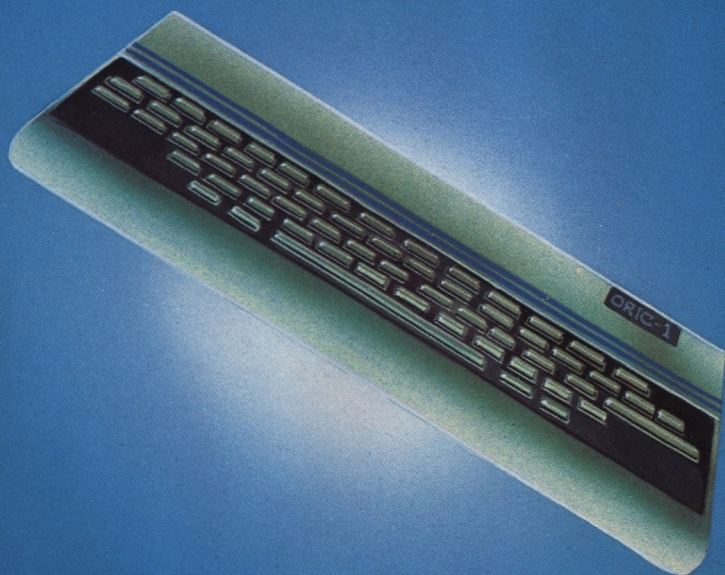


Personal
Computer
World
CENTURY

THE

ORIC

HANDBOOK



Peter Lupton &
Frazer Robinson

Includes full details
of the new
ORIC
ROM

THE ORIC HANDBOOK

PETER LUPTON & FRAZER ROBINSON



CENTURY COMMUNICATIONS

LONDON

Copyright © Peter Lupton and Frazer Robinson 1983

All rights reserved

First published in Great Britain in 1983
by Century Communications Limited,
76 Old Compton Street,
London W1V 5PA

ISBN 0 7126 0175 9

Printed in Great Britain in 1983 by
The Garden City Press Ltd.,
Letchworth,
Hertfordshire.

Typeset by Spokesman, Bracknell

CONTENTS

Acknowledgements

Introduction

1	Computers and Programs	1
2	Setting up your ORIC	3
3	Communication	9
4	Programming	16
5	Program Control	32
6	Data and Programs	50
7	Pieces of Strings	59
8	Functions	69
9	Logical Thinking	78
10	Making a Noise	85
11	Characters	103
12	Low Resolution Graphics	120
13	High Resolution Graphics	131
14	Loading and Saving Programs	148
15	Advanced Programming	154
16	Example Programs	166
17	The New Operating System	188

APPENDICES

1	BASIC Commands	207
---	----------------	-----

2	BASIC Error Messages	224
3	Attributes	229
4	ASCII Codes	230
5	Control Codes	231
6	Screen Memory Maps	232
7	System Memory Map	234
8	Numbering Systems	235
9	Musical Notation	240
10	Speeding up Programs	244
11	Editing BASIC Programs	246
	Index	248

ACKNOWLEDGEMENTS

Our thanks to all those who in some way contributed to this book, including friends, families and colleagues whose support (or lack of it !) kept us going.

Special thanks to Don Gray and Bob Dinsdale, without whose help this book would never have been finished, and to Richard Gollner and Mike Lord for their efforts on our behalf. Thanks also to Dr. Paul Johnson of Oric Products International Ltd. for assistance with the new ROMs.

Finally, HELLO! to all those people who have been deprived of our company for such a long time, especially to Anne, and to Simon and Angela.

INTRODUCTION

This book is an introduction to the ORIC-1 computer and its facilities. It takes you from the first steps in BASIC programming, pointing out the pitfalls and explaining some 'Tricks of the trade' through to a detailed understanding of how to write good, structured programs, with many examples along the way.

Separate chapters explain how to make the most of the ORIC's remarkable sound and graphics facilities - again, many example programs are given.

The extra facilities of the new Version 1.1 Operating System are fully described. These facilities greatly increase the potential of an already powerful machine.

A set of useful appendices ensures that the book will remain a valuable reference work long after you have grasped the principles of the machine.

We hope that you find the book both enjoyable and instructive, as you discover the full potential of the ORIC-1.

Peter Lupton and Frazer Robinson 1983

CHAPTER 1

COMPUTERS AND PROGRAMS

Despite everything you may have heard about computers being hyperintelligent machines which are on the point of taking over the world, a computer is not really intelligent at all. A computer is at heart little more than a high-speed adding machine, similar to an electronic calculator, but with more sophisticated display and memory facilities.

The feature that gives computers their immense flexibility and potential is this: they can store lists of instructions which they can read through and obey at very high speed. These lists of instructions are called programs, and most of this book is concerned with how these programs are written.

The computer instructions which form programs must follow certain rules, or the computer will not be able to understand them. The rules for writing programs resemble the rules of a spoken language, and so the set of instructions is often said to form a programming language. There are many different computer programming languages; the one that the ORIC understands (in common with most other personal computers) is called BASIC. (The name BASIC is an acronym for Beginners' All-purpose Symbolic Instruction Code.)

A programming language is much simpler than a human language because computers, for all their power, are not as good at understanding languages as people are. The BASIC language used by the ORIC has less than 120 words. The rules for combining the words - the 'grammar' of the

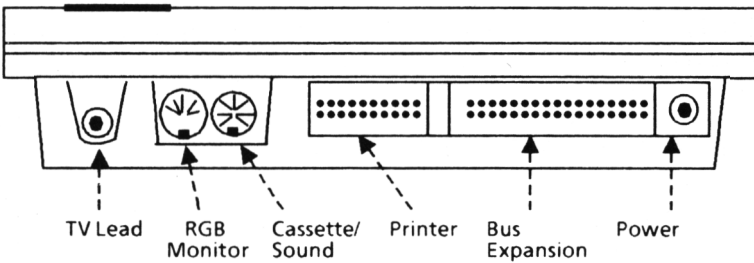
language - are much more strict than for a language like English, again because it is difficult to make computers that can use languages in the relaxed sort of way in which we speak English. These may seem like limitations, but in fact as you will discover BASIC is still a powerful language, and it is possible to write programs to perform very complex tasks.

Finally, remember this. Your ORIC will not do anything unless you tell it to, so whatever happens, you're the boss. The ORIC won't take over the world unless you make it!

CHAPTER 2

SETTING UP YOUR ORIC

Before connecting anything to your ORIC, look at the back of the computer and compare it with this diagram.



There are several sockets, through which the ORIC passes and receives information, and one through which it gets the electrical power it needs to operate. The sockets are not labelled, so be sure to refer to the diagram before plugging anything in.

DISPLAY

The ORIC uses a standard domestic T.V. to communicate with you, and for this almost any T.V. will do. To get the best results, use a modern, good quality colour T.V. If you use a black and white set, the colour displays produced by the ORIC will appear as shades of grey.

To connect the ORIC to the T.V., plug in the aerial lead supplied to the aerial socket of the T.V., and

plug the other end into the socket at the back of the ORIC (check the diagram). The lead has a different type of plug at each end, so take care that you don't try to force in the wrong one.

POWER

The ORIC needs a low voltage D.C. supply, and this is obtained from the large black 'plug' supplied with your computer. Plug this power supply into the mains, and plug the thin black lead into the ORIC. The socket for the power supply is at the opposite end of the machine to the one for the T.V. lead.

CASSETTE RECORDER

The ORIC, like most other small computers, uses an ordinary cassette recorder to save programs or information, so that you don't have to type them in every time you need them.

Surprisingly, the ORIC works best with cheap portable mono recorders and may not work at all with your hi-fi model. This is because expensive recorders contain circuitry which modifies the sound before it is recorded. This is fine for music, but not for computer programs.

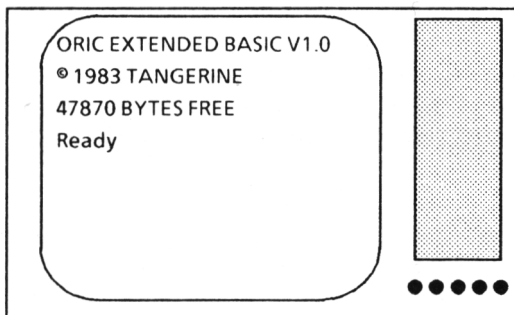
The cassette socket is shown on the diagram. To connect up your ORIC, plug one end of the cassette lead into the computer, and the other into the PLAY/RECORD socket on the cassette recorder.

You will find details of how to use the cassette recorder to load and save programs in Chapter 14.

TUNING

To get a display to appear on the T.V. screen, tune to Channel 36, or, on a pushbutton set, use a spare

channel and keep tuning until you see this appear on the screen:



If you see a pattern of squares and lines, pull out the power lead from the ORIC, wait a few moments, and plug it back in again. This is the best way to switch the ORIC on and off, rather than at the mains.

If you can't get the correct display to appear, refer to the Trouble Shooting section, but note that it will take a few seconds from switching on for the screen to clear, and the message to appear.

Assuming that all is well, and you do get a display, then you are ready to start learning.

TROUBLE SHOOTING

If you've had problems getting a display on your T.V., and you can't find the answer in this section, then you must refer to your dealer. DON'T be tempted to open the case, as you'll not improve matters and will invalidate your guarantee.

No Display At All

Check that the aerial lead is connected.

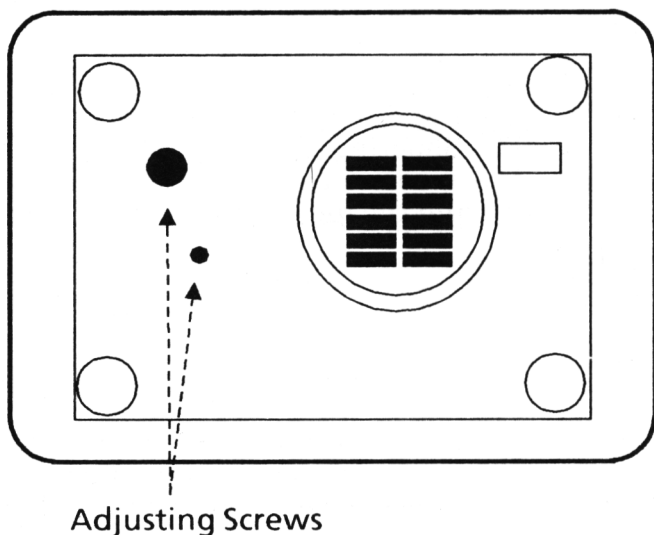
Check the tuning of the T.V., and if possible try a different T.V.

Check that power is getting to the ORIC - you should be able to hear a faint humming noise coming from the computer if it is switched on.

Faint or Unstable Display

If you have some kind of display, and no amount of tuning will improve it, there are two adjustments you can make to achieve the best possible display.

If you turn the ORIC over, you will find two small, circular holes close to one edge. The diagram shows where they are.



You will need a very small screwdriver and a strong light to see inside the holes.

Beneath the larger hole there is a small slot. Rotating this will tune the computer to give a stable picture.

Keeping an eye on the screen, rotate the slot with the screwdriver in both directions to find the position which gives the clearest and most flicker free display.

Now insert the screwdriver into the brass screw beneath the smaller hole. Rotate the screw until you find the best display.

These adjustments are best carried out when the T.V. and computer have been switched on for a while, so that they have both reached their operating temperature.

When you know more about how the ORIC works, you may like to return to this section to get the best colours.

With a little time and careful tuning, it is possible to get a clear and stable display on nearly all types of T.V. If you are unsuccessful, consult your dealer.

If you are thinking of buying a T.V. especially for use with your ORIC, it's worthwhile taking the computer to the shop, as certain types of T.V. seem to give better results than others.

For the best quality display, you could buy a monitor. This is a display specifically designed for use with a computer, and contains no circuitry for T.V. reception. However, a good colour monitor can cost two or three times the price of an ORIC-1!

The ORIC provides a standard output for a monitor, through the socket next to the cassette recorder socket. Your dealer will advise you on the connection.

CHAPTER 3

COMMUNICATION

Before you can write programs for your ORIC, you must find out how to communicate with it.

Communication is a two-way process: you must give the ORIC instructions, and you must be able to find out how it responds to them. You give instructions using the keyboard, and the ORIC displays its response on the TV screen.

Type the following phrase on the keyboard.

```
PRINT "ORIC-1"
```

You will see the letters appear on the screen as you type them, and hear a 'click' as you press each key. The flashing square - called the cursor - will move to indicate where the next letter you type will appear. Nothing else happens though - the ORIC has not yet obeyed your instruction.

Now press the RETURN key. The words 'ORIC-1' appear on the screen, and the word 'Ready' is printed to tell you that the ORIC is waiting for your next command. (If instead of printing 'ORIC-1' the computer prints '?SYNTAX ERROR' or 0, it means you have made a typing mistake. Try again!)

So, to give the ORIC a command you type it at the keyboard and press RETURN. Try another command:

```
PRINT "HELLO!"
```

Again, the letters between the quotation marks are printed. You can tell the ORIC to print any sequence of letters, but you must remember to put them between quotation marks. Try making it print your name.

You don't have to type PRINT in full every time - the question mark has the same meaning to the ORIC. Try

? "HELLO!"

(remembering to press RETURN, of course).

MANAGING THE DISPLAY

As well as typing on to the screen, there are a number of different ways to alter the display.

The Cursor

This can be moved around the screen using the keys marked with arrows at either end of the space bar. Try moving the cursor around, and watch what happens when it gets to the edge of the picture: when it reaches one side it reappears at the other side, but when the cursor reaches the top or bottom of the screen the picture starts to move in the opposite direction. This vertical movement is called **scrolling**. Notice that anything that disappears at the top or bottom when the picture scrolls cannot be recovered by scrolling the other way: it is lost for ever.

Corrections

If you notice that you have made a typing mistake before you press RETURN, you can correct the error by using the DEL key. For example, if you type

PRONT "ORIC"

and then realise your mistake, press DEL until the first O is erased and then retype the rest of the line.

There is another way of making corrections, or editing what you have typed, which is more convenient for major corrections than retyping. Details of this are given in Appendix 11.

NOTE You can **not** make corrections by moving the cursor over the error with the arrowed keys and then retyping. The ORIC will not understand what you have done and will display the '?SYNTAX ERROR' message.

Clearing the screen

This can be done in two ways. You can either hold down the CTRL key and press L, or type CLS and press RETURN.

NUMBERS

As well as printing words, the ORIC can also handle numbers. Try

PRINT 5 (RETURN)

You can do arithmetic.

Addition:

PRINT 5+3

Subtraction:

PRINT 7-4

Multiplication:

```
PRINT 3*5
```

Division:

```
PRINT 15/6
```

Powers(exponentiation):

```
PRINT 3 ↑ 2
```

You can ask the ORIC to calculate longer expressions, such as:

```
PRINT 3*5 + (5-3)/(2+1) * 7/8
```

In working out expressions like this, the ORIC follows strict rules about the order in which the various arithmetical operations are performed.

- | | |
|---------------|--|
| First | The expressions inside brackets are evaluated. |
| Second | Any powers (squares, cubes, etc.) indicated by ↑ are worked out. |
| Third | The ORIC performs the multiplications and divisions. |
| Fourth | The additions and subtractions are performed. |

If you are unsure, put brackets round the bit you want calculated first. Try the following examples, and see if you can work out the answers yourself to check that you know what the ORIC is doing.

```
PRINT 2 * 2 + 2 * 2
PRINT 2 + 2 * 2 + 2
PRINT 6 + 3 / 5 - 4
PRINT 3 ↑ 2 + 4 ↑ 3/2
PRINT (3 + 4) * 2
```

COLOUR

The colours of the writing on the screen and of the background can be changed. The foreground colour (the colour of the writing) is altered by the command **INK** and the number of the desired colour. Try this:

INK 1

The writing on the screen will turn red.

The background is changed in a similar way by the **PAPER** command.

PAPER 3

will turn the background yellow.

There are eight colours, and their numbers are as follows:

BLACK	0
RED	1
GREEN	2
YELLOW	3
BLUE	4
MAGENTA	5
CYAN	6
WHITE	7

Try changing the foreground and background colours to test the possibilities. If all the writing on the screen disappears, don't panic; you will have set the **INK** and **PAPER** to the same colour. The ORIC will still understand commands, even if you can't see what you are typing.

SOUNDS

There are four special sounds which the ORIC will make in response to your commands. These are:

PING
SHOOT
ZAP
EXPLODE

To hear a sound, type the name and press RETURN.

MULTIPLE COMMANDS

You can put more than one instruction in one command if you separate the instructions with colons (:). Try this:

```
CLS: PRINT "3 + 4 = "3+4: PING
```

Or this:

```
CLS: PAPER 3: INK 4: ZAP: PRINT:  
PRINT "BANG!": EXPLODE
```

It doesn't matter if you run over the end of the line on the screen, as long as there are no more than 78 characters altogether in the command (including the spaces). The ORIC will warn you by making the 'PING' sound when you get near the limit.

Entering a number of instructions together like this can get very cumbersome. In the next chapter we will discover a way of giving the computer a very large number of instructions all at once - the **PROGRAM**.

SUMMARY

PRINT tells the ORIC to print something.

```
PRINT "ABCDEFGG"
```

prints the characters between the quotation marks.

```
PRINT 3+4
```

prints the result of the sum.

These can be combined:

```
PRINT "3+4*5+7 = " 3+4*5+7
```

SOUNDS	ZAP
	SHOOT
	PING
	EXPLODE

COLOURS	INK changes the foreground,
	PAPER changes the background.

Both **INK** and **PAPER** must be followed by a number between 0 and 7.

MULTIPLE COMMANDS must be separated by colons (:).

CHAPTER 4

PROGRAMMING

At the end of Chapter 3, we saw that it is possible to give the ORIC a number of instructions at one time by writing them one after another on one line. We will now look at a much more powerful way of doing this - the **program**.

A computer program is a numbered list of computer instructions which are entered together and stored by the computer to be obeyed later. Let's look at the last example of Chapter 3 and see how we can turn it into a program. In Chapter 3 we had:

```
CLS: PAPER 3: INK 4: ZAP: PRINT:  
PRINT "BANG!": EXPLODE
```

Rewritten as a program it looks like this:

```
10 CLS  
20 PAPER 3  
30 INK 4  
40 ZAP  
50 PRINT  
60 PRINT "BANG!"  
70 EXPLODE
```

Type in each line in turn (remembering to press RETURN after each). You will see that the ORIC does not obey the instructions as they are entered, and that the 'READY' message does not appear after you press RETURN. The ORIC recognises that each line is part of a program, because each begins with a number, and the program is stored for future use.

When you have typed in all the lines, you can inspect the stored program by typing **LIST** (and pressing **RETURN**, of course). The instructions are listed in numerical order on the screen. You can list parts of the program by specifying line numbers. For example:

LIST 30-50	lists all lines from 30 to 50.
LIST 20	lists only line 20
LIST - 30	lists all lines up to and including line 30
LIST 30 -	lists all lines from line 30 to the end of the program

To order the computer to act on the program instructions use the command **RUN** (followed by **RETURN**). This tells the ORIC to read through the stored program and obey each instruction in order. You will see that the program gives exactly the same results as the multiple instructions in Chapter 3. (If it doesn't, and the ORIC prints a message such as 'SYNTAX ERROR IN LINE 20', it means there is a typing mistake in that line and the ORIC cannot understand the instruction. **LIST** the line to check, and retype it.)

A program can be **RUN** as many times as you like. It will be stored in the computer's memory until you switch it off, unless you alter or delete the program.

It is possible to begin execution of a program from a line other than the first by specifying the line number. For example **RUN 20** begins the program at line 20, and line 10, the **CLS** instruction, is ignored.

Programs can be altered or added to at will. To alter an instruction, retype the line. If you enter:

20 PAPER 1

the original line 20 will be replaced by the new one, and the next time you **RUN** the program the screen will turn red instead of yellow. (To restore the black and white screen after running the program you can type

PAPER 7: INK 0

and press RETURN.)

Extra lines are added by typing them as before:

35 ZAP: ZAP: ZAP

makes the program even noisier. Notice that the ORIC inserts the extra line in the appropriate place: it is **not** put at the end.

NOTE It is a good idea always to number program lines in steps of 10 to allow extra or forgotten lines to be inserted without having to renumber the whole program.

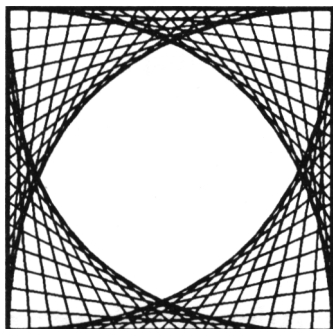
To delete a line, type the line number and press RETURN. The line is deleted from the list in the ORIC's memory.

Now see if you can alter the example program to give different **INK** and **PAPER** colours, and to **PRINT** different words on the screen.

When you have exhausted the possibilities of this program, you can use the command **NEW** to delete the whole program. You will then be ready for the next program.

PRETTY PATTERNS

Here is a more interesting program for you to try. The program draws a pattern like the one shown in the diagram below, using the ORIC's high resolution graphics facilities. Many of the commands used will be unfamiliar to you at present, but you need not worry about that. Just type in the program and watch what it does.



```
10  TEXT
20  PAPER6: INK4
30  HIRES
40  PAPER6: INK4
50  PRINT CHR$(17)

100 CURSET 20,0,1
110 DRAW 200,0,1
120 DRAW 0,199,1
130 DRAW -200,0,1
140 DRAW 0,-199,1

200 FOR X=0 TO 199 STEP 3
210 CURSET 21,X,3
220 DRAW X, 199-X, 1
230 DRAW 199-X, -X, 1
240 DRAW -X, X-199, 1
250 DRAW X-200, X, 1
```

```
260  NEXT X  
  
300  PRINT CHR$(17)  
310  END
```

Check the program carefully for typing errors, and then **RUN** it. If the program stops and the message '? SYNTAX ERROR IN LINE xxx' is printed, it means you have made a typing mistake in that line. If any other error message is printed, check the whole of the program for mistakes.

To restore the normal display after running the program, type **TEXT** and press **RETURN**. The screen can not be cleared by **CLS** or **CTRL** and **L** because the pattern is drawn in high resolution mode. This will all be made clear in Chapter 13, so don't worry about it at this stage.

ARITHMETIC IN PROGRAMS - VARIABLES

We discovered in Chapter 3 that the ORIC will print the answers to arithmetical problems in response to commands such as

```
PRINT 3+5
```

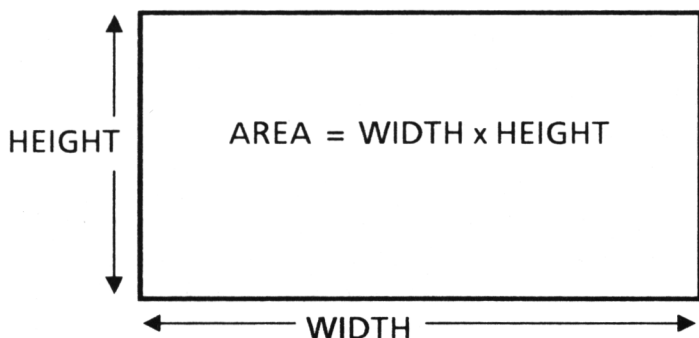
This sort of instruction can be included in a program like this:

```
10  CLS  
20  PRINT "3+5 = " 3+5
```

but it would be difficult to make much use of this for working out your personal finances.

What gives a computer the power to perform complex data processing (or 'number crunching') tasks is its ability to do algebra: to use names as symbols for numbers. This means that programs

can be written with names to symbolise numbers, and then **RUN** with different numbers attached to the names. These number-names are called **VARIABLES** because they represent varying numbers. Let's try an example.



The area of a rectangle is equal to the width of the rectangle multiplied by the height. By using names instead of numbers we can write a program to work out the area of any rectangle. For example:

```
10  WIDTH = 8
20  HEIGHT = 12
30  AREA = WIDTH * HEIGHT
40  PRINT "AREA = " AREA
```

This program, when **RUN**, will print the area of a rectangle 12 inches by 8 inches. By changing the numbers in lines 10 and 20 we can obtain the area of a rectangle of any size.

VARIABLE NAMES

Any number of letters can be used in a variable name, but only the first two characters of the name are considered by the ORIC; the rest are ignored. This means for example that if you use the names

FR
FRUIT
FRIEDEGG

the computer will treat them as the same variable.

All variable names must begin with a letter, but the other characters may be numbers, for example A1, EGG7, LAST1. There must not be any spaces in a variables name: this will confuse the ORIC.

NOTE You cannot use as a variable name any of the BASIC command words, or any name which includes a command word. If you do, your program will not run, but will give SYNTAX ERROR messages wherever the name is used. For example, LENGTH contains LEN, and BREADTH contains READ, both of which are BASIC words. (See Appendix 1 for a full list of all the BASIC reserved words.)

TYPES OF VARIABLE

There are three different types of variable, two of which represent numbers, the third representing words. The types which represent numbers are Real and Integer variables; the type representing words are called String variables because they contain sequences or 'strings' of characters. The variables in the program above are real variables.

Real Variables

These are used to represent numbers and can have any value, whole numbers or fractions. Real variables should be used in all arithmetical programs.

Integer Variables

These can only represent whole numbers (or integers), not fractions. They may be used when counting events or objects, or to store constant integer values. For example, a program which kept a record of the number of chocolate bars in stock at a shop could use integer variables to count them, but the takings of the shop would have to be stored in real variables, or fractions of pounds (i.e. pennies) would be ignored.

Integer variables must be identified by the % sign after the variable name, as in CHOCBARS%.

So numbers can be represented in two ways: by real variables and integer variables. Real variables can have any value. For example:

```
REAL = 3.72
SIZE = 87.3 * 2.5
```

are both real variables. Integer variables must be whole numbers:

```
NUMBER% = 3+6
COUNT% = COUNT% + 1
CHOCBARS% = 1000000
```

If you try to give an integer variable a fractional value only the whole number part is stored. So

```
10 NUMBER% = 7.5
20 PRINT NUMBER%
```

will print the value 7.

String Variables

A string variable is used to store words or letters. The variable name for a string variable must be

followed by a dollar sign (\$) to distinguish it from the other types of variable. Examples are:

```
NAME$ = "WINSTON CHURCHILL"  
STRING$ = "ABCDEFGG"
```

Note that the letters defining the variable contents are enclosed by quotation marks.

The maximum number of characters a string variable can hold is 255. If you try to make a string longer than that the ORIC will display the message 'STRING TOO LONG ERROR'.

NOTE It is possible to have variables of different types with the same name, such as:

```
NAME = 87.76  
NAME% = 3  
NAME$ = "GEORGE WASHINGTON"
```

The computer will not be confused, but you might, so be careful!

ARRAYS

Each of the three types of BASIC variable, real, integer and string, may be used in an array form. In normal use a variable name represents one stored number or string. In an array, the variable name represents a collection of stored information, with one or more reference numbers identifying the individual items. An array can be pictured as a collection of boxes. Here is a diagram of an array which uses one reference number.

The array is a string array and has 6 elements numbered from 0 to 5. The array would be used like this:

REFERENCE NUMBER	CONTENTS
0	LUCY
1	JULIE
2	MIKE
3	DAVE
4	CAROL
5	SUE

Array NAME\$

```
10  NAME$(0) = "LUCY"  
20  NAME$(1) = "JULIE"  
30  NAME$(2) = "MIKE"  
40  NAME$(3) = "DAVE"  
50  NAME$(4) = "CAROL"  
60  NAME$(5) = "SUE"
```

```
100  FOR N = 0 TO 5  
110  PRINT N, NAME$(N)  
120  NEXT N
```

As mentioned above, an array may have more than one reference number. The number of reference numbers per item is called the number of **dimensions** of the array. Here is a program using an array with two reference numbers per item:

```
10  DIM P$(2,2)  
20  P$(0,0) = "FLORENCE"  
30  P$(0,1) = "NIGHTINGALE"  
40  P$(1,0) = "GEORGE"  
50  P$(1,1) = "WASHINGTON"
```



```

60   P$(2,0) = "ISAAC"
70   P$(2,1) = "NEWTON"

100  FOR A = 0 TO 2
110  FOR B = 0 TO 1
120  PRINT P$(A,B); "  ";
130  NEXT B
140  PRINT
150  NEXT A

```

Here again the array can be thought of as a collection of storage boxes, this time laid out in a grid as shown below.

	B = 0	B = 1
A = 0	FLORENCE	NIGHTINGALE
A = 1	GEORGE	WASHINGTON
A = 2	ISAAC	NEWTON

Array P\$

You can use arrays with more than two dimensions, but you will find the ORIC's memory will not hold arrays of more than about four dimensions (the exact number depends on the number of elements in each dimension). The maximum allowed value of each reference number may be anything up to several thousand, but this depends on how much memory is occupied by the program itself, and by the other variables of the program.

A **DIM** command must be included at the beginning of any program which will use arrays of more than one dimension, or with more than eleven elements (0-10). The command tells the ORIC to reserve memory space for the array. You do not

need a **DIM** command if you are using one-dimensional arrays with no more than eleven elements numbered from 0 to 10, as the ORIC is able to handle these automatically.

GETTING VALUES INTO PROGRAMS

It would be very inconvenient to have to alter a program to make it handle different numbers, so there are a number of instructions which allow numbers or letters to be given to a program while it is running. The first of these is **INPUT**.

When the computer finds an **INPUT** statement in a program, a question mark is displayed on the screen. The computer waits for you to type in a number or letter string, which it will then store in a variable before continuing with the rest of the program. Type **NEW (RETURN)** and then try this example:

```
10 INPUT NUMBER
20 PRINT NUMBER
```

When you **RUN** this program you will see a question mark on the screen. Type a number and press **RETURN**: the number is printed.

NOTE If you typed anything other than a number the computer would print the message 'REDO FROM START'. It would then display another question mark and wait for you to type the number in again. This is because the variable **NUMBER** is a real variable and can only store numbers. If you press **RETURN** without typing anything, another question mark is displayed on the next line and the

ORIC continues to wait for you to enter a number.

The **INPUT** instruction can handle strings too, if you specify a string variable in the **INPUT** command:

```
10  INPUT NAME$  
20  PRINT NAME$
```

This program will accept both numbers and letters and store them in a string variable.

Messages can be printed before the question mark to tell the person using the program what to type in. Change line 10 to:

```
10  INPUT "TYPE A WORD"; NAME$
```

and **RUN** the program again. You can use any message you like, but it must be contained within quotation marks, and there must always be a semicolon between the message and the variable which will store the number.

It is possible to use one **INPUT** command to input two or more numbers or strings.

```
10  INPUT "NAME, AGE"; NAME$, AGE  
20  PRINT NAME$  
30  PRINT AGE
```

The variables must have commas separating them in the **INPUT** command; and when you type in the information you must use a comma to separate the items. If you press **RETURN** before entering all the items the ORIC will print two question marks and wait for the remaining items. If you enter too many items the surplus ones will be ignored and the ORIC will print 'EXTRA IGNORED'.

Let's use what we know about variables and **INPUT** to improve the area program. Clear the ORIC memory by typing **NEW (RETURN)**, and then type in this program.

```
10  REM  IMPROVED AREA PROGRAM
20  CLS
30  INPUT "ENTER WIDTH"; WIDTH
40  INPUT "ENTER HEIGHT"; HEIGHT
50  AREA = WIDTH * HEIGHT 'FIND
    AREA
60  PRINT: PRINT "AREA = " AREA
```

This program will read the two numbers you type in for width and height and print the area of the rectangle. Using **INPUT** has made the program much more flexible: we don't have to alter the program to use different numbers.

REMARKS

Line 10 of the last program is a **remark** or **comment** statement. These are used to hold comments, notes and titles to make the purpose of a program and the way in which it works clear to someone reading the listing. Remarks are identified by the word **REM** before the remark and have no effect when the program is **RUN**.

Comments may be added to the ends of program lines, as in line 50. The apostrophe (') indicates the end of the instruction and the beginning of the comment; everything after the apostrophe is ignored by the ORIC when running the program.

You should always put plenty of remarks in programs, because although you may understand how a program works when you write it, three months later you will have forgotten. If you need to modify a program at some time after writing it,

remarks will make it much easier to remember how the program works.

SUMMARY

PROGRAMS

A **program** is a sequence of computer instructions, identified by numbers.

LIST displays the lines of a program.

RUN starts the execution of a program.

NEW deletes a program from the computer's memory.

VARIABLES

Variables represent numbers and words in programs.

There are three types of variable:

Real representing any numbers.

Integer representing whole numbers, and distinguished by % at the end of the variable name.

String representing words, and distinguished by \$ after the name.

INPUT is used to enter numbers or words into variables while a program is running.

Variables may be collected in arrays, with the same variable name, but distinguishing reference numbers.

REMARKS

REM is used to put remarks into programs for the programmer's benefit.

Comments may be added to the ends of program lines following an apostrophe (').

CHAPTER 5

PROGRAM CONTROL

In the last chapter we defined a program as a numbered list of instructions to the computer, which are obeyed in order from beginning to end. In this chapter we will find out how to write programs in which some instructions are executed more than once. This makes programs more efficient, as instructions which must be repeated need to be written only once. We will also discover that the computer can make decisions.

REPETITION

A section of program can be repeated many times using the two instructions **REPEAT** and **UNTIL**.

Try this example:

```
10  COUNT = 0
20  REPEAT
30  COUNT = COUNT + 1
40  PRINT COUNT
50  UNTIL COUNT = 10
```

which prints the numbers from 1 to 10 on the screen.

The **REPEAT** and **UNTIL** instructions are used like this:

REPEAT is placed at the beginning of the section of program to be repeated.

UNTIL is placed at the end of the repeated section, and is followed by a condition which must be satisfied if the section is **not** to be repeated.

The example program works like this.

Line 10 sets **COUNT** to zero. This instruction is only obeyed once, because it is not between the **REPEAT** and **UNTIL** commands.

Line 20 **REPEAT** marks the beginning of the repeated section.

Lines 30 and 40 add one to count and print the new value.

Line 50 marks the end of the repeated section. When this line is executed the computer returns to line 20 (the **REPEAT** command) unless the condition after **UNTIL** is met, i.e. that **COUNT** is equal to 10.

Here is another example:

```
10  REPEAT
20  INPUT "ENTER A NUMBER"; NUMBER
30  UNTIL NUMBER < 10
40  ZAP
```

This program repeats the **INPUT** command until a number less than 10 is entered. This is a useful technique in longer programs for checking that numbers typed in are within desired limits.

CONDITIONS

The condition for continuing the program which follow the **UNTIL** command may be one of many possible alternatives. Examples are:

UNTIL COUNT = 10

Continue when variable (COUNT) equals a number.

UNTIL TEST < 100

Continue when variable less than a number.

UNTIL COUNT > NUMBER

Continue when variable greater than a number.

UNTIL NUMBER < > VALUE

One variable not equal to another (greater than or less than).

UNTIL X >= Y

Greater than or equal to.

UNTIL X <= Y

Less than or equal to.

Two conditions may be combined, as in:

UNTIL X = 1 OR X = 2

UNTIL A = 3 AND NAME\$ = "ORIC"

In all of these the items being compared may both be variables, or one may be a variable and the other a number. (There's not much point in comparing two numbers!) For further details see Chapter 9, Logical Thinking.

The wide variety of possible conditions controlling a **REPEAT ... UNTIL** sequence makes it a very powerful programming tool.

To sum up, **REPEAT** and **UNTIL** are used to repeat a section of program until the conditions given after **UNTIL** are met. This allows calculations to be refined until a suitably accurate answer is obtained and is useful for testing inputs to check that they are within the necessary limits.

Repeating sections of program are generally referred to as **loops**. As well as the **REPEAT** and **UNTIL** commands for controlling loops the ORIC has a second set of commands: **FOR** and **NEXT**.

FOR ... NEXT LOOPS

This type of loop differs from a **REPEAT ... UNTIL** loop in two ways.

- 1 The loop repeats for a set number of times.
- 2 A variable is automatically altered on each repetition.

Look back at the first example of a **REPEAT ... UNTIL** loop earlier in this chapter which printed the numbers from one to ten. Using **FOR ... NEXT** the program can be written like this:

```
10    FOR COUNT = 1 TO 10
20    PRINT COUNT
30    NEXT COUNT
```

The section of program between the **FOR** and the **NEXT** commands is repeated for each value of **COUNT**, and **COUNT** is automatically increased by one every time the **NEXT** command is met.

The variable in a **FOR ... NEXT** loop need not be increased by 1; any increase or decrease can be specified using the command **STEP**. Try changing line 10 to:

```
10    FOR COUNT = 0 TO 30 STEP 3
```

and **RUN**ning the program again. **COUNT** is now increased by 3 each time the loop is repeated. The **STEP** can be negative so that the numbers get smaller:

```
10    FOR COUNT = 50 TO 0 STEP -2.5
```

It is not necessary to specify the variable after **NEXT**;

```
30    NEXT
```

would work just as well. However when you write longer programs you will find that they are clearer if the variables are always specified. The ORIC will not be confused, but you will be as you write the program!

To recap, the instructions are used like this:

```
FOR V = X TO Y STEP Z
```

is used to begin a loop (**V** is a variable and **X**, **Y** and **Z** may be variables or numbers), and

```
NEXT V
```

marks the end of the loop.

The variables used in **FOR ... NEXT** loops may be real or integer variables, but **not** string variables, and **not** elements of arrays.

NESTED LOOPS

Both **REPEAT ... UNTIL** and **FOR ... NEXT** loops can be **nested** - that is, one loop can contain one or more other loops. Here are two examples:

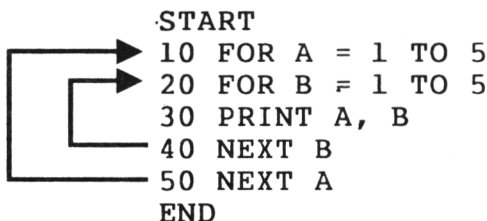
```
1  10    A = 0
    20    REPEAT
    30      : A = A+1
    40      : B = 0
    50      : REPEAT
    60        : B = B+1
    70        : PRINT A, B
    80      : UNTIL B = 4
    90    UNTIL A = 3

2  10    FOR A = 1 TO 3
    20      : FOR B = 1 TO 4
    30        : PRINT A, B
    40      : NEXT B
    50    NEXT A
```

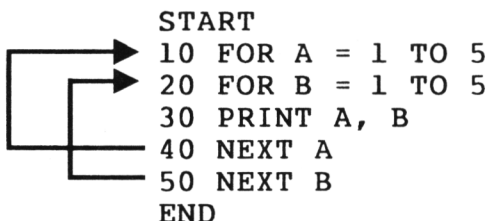
The two programs give the same results but using different types of loop.

Notice that the instructions within the loops have been indented by placing a colon at the beginning of the line and typing spaces. This makes no difference to the working of the program, but it makes it easier for you to see where the loops are when reading the program.

For any nested loops, it is **very important** that the enclosed loop is **completely within** the outer loop, otherwise the program **will not work**. To clarify this, let's examine the order in which instructions are obeyed in program 2.



The two loops are arranged so that one is completely within the other. If lines 40 and 50 were swapped over, the result would be:



You can see that the two loops overlap and the order in which the instructions should be obeyed is not clear.

This rule applies to all types of loop, **REPEAT ... UNTIL** as well as **FOR ... NEXT**: the loops must **always** be one within the next.

The maximum possible number of **FOR ... NEXT** loops which can be nested one within the next is 10; the maximum number of **REPEAT ... UNTIL** loops which can be nested depends on the amount of memory available. If you write a program with too many loops nested within each other, when it is **RUN** the ORIC will print the message 'OUT OF MEMORY ERROR'.

NOTE Where two or more nested **FOR ... NEXT** loops have their **NEXT** commands one after the other they may be combined by specifying both or all the variables after one **NEXT**. Program 2 would therefore end

```
40 NEXT B, A
```

Again the order of the variables in line 40 is the reverse of the order in which the corresponding **FOR** commands occur.

JUMPS

The computer can be instructed to jump from one program line to another using the command **GOTO**. When the program

```
10    GOTO 30
20    ZAP
30    PING
```

is run the **ZAP** is not heard because the **GOTO** instruction in line 10 sends the ORIC straight to line 30. (Note that **GOTO** must be typed without a space in the middle - do not type **GO TO**.)

The jump can be to a lower numbered line. The following program will go on for ever, unless you stop it by holding down **CTRL** and pressing **C**.

```
10    ZAP
20    GOTO 10
```

This is called an endless loop, and is to be avoided!

It is possible to use a variable to hold the destination line number, as in this program.

```
10    A = 40
20    GOTO A
30    ZAP
40    PING
```

The program jumps from line 20 to line 40, as A contains 40.

WAITS

The command **WAIT** tells the ORIC to pause while running a program. The command is used like this:

```
WAIT T
```

where T is a number or variable with a value between 0 and 65535. The number specifies the length of the pause in hundredths of a second, so

```
WAIT 100
```

gives a one second pause;

```
WAIT 6000
```

a one minute pause, and

```
WAIT 65535
```

a pause of nearly 11 minutes.

Try increasing the length of the **WAIT** in line 30 of this program to change the sound from machine-gun fire to single shots.

```
10    FOR S = 1 TO 20
```

```
20    SHOOT
30    WAIT 15
40    NEXT S
```

DECISIONS

The commands **IF**, **THEN** and **ELSE** are used in programs to make decisions. A variable is tested, and one of two alternative actions is taken, depending on the result of the test.

```
10    FOR I = 1 TO 10
20    PRINT I
30    IF I = 5 THEN SHOOT ELSE PING
40    WAIT 20
50    NEXT I
```

The format of **IF ... THEN ... ELSE** is:

IF (condition) **THEN** (instructions) **ELSE**
(instructions)

The condition after **IF** varies in the same way as the condition after an **UNTIL**. The instructions after **THEN** are carried out if the condition is met; the instructions after **ELSE** are carried out if the condition is not met.

The **ELSE** and its instructions can be omitted, as in

```
10    FOR P = 1 TO 20
20    PRINT P
30    IF P = 10 THEN PING
40    WAIT 30
50    NEXT P
```

IF ... THEN ... ELSE can be used to control jumps:


```
10    INPUT "WHAT NUMBER AM I  
      THINKING OF"; N  
20    IF N < > 3 THEN PRINT  
      "WRONG":GOTO 10  
30    PRINT "CORRECT!"
```

If the jump is the only instruction after **THEN** or **ELSE**, the word **GOTO** can be omitted:

```
10    INPUT "WHAT AM I"; A$  
20    IF A$ = "ORIC" THEN 60  
30    PRINT: PRINT A$"? NO, TRY  
      AGAIN"  
40    PRINT  
50    GOTO 10  
60    PRINT: PRINT"GOOD GUESS!"
```

EXAMPLE PROGRAM - SORTING NUMBERS

As an example of what can be done using loops and decisions, here is a program which sorts ten numbers into ascending order.

```
10    REM    SORTING PROGRAM  
20    DIM NUM(10): DIM S(10)  
30    REM    INPUT 10 NUMBERS  
40    CLS: PRINT  
50    FOR N = 1 TO 10  
60      : INPUT "ENTER NUMBER";  
        NUM(N)  
70    NEXT N  
100   REM    COPY NUMBERS TO ARRAY S  
110   FOR C = 1 TO 10  
120     : S(C) = NUM(C)  
130   NEXT C  
200   REM    SORT NUMBERS  
210   REPEAT  
220     : COUNT = 0  
230     : FOR N = 1 TO 9
```

```
240 :      IF S(N+1) >= S(N) THEN
290
250 :      TEMP = S(N+1)      'SWAP
      NUMBERS
260 :      S(N+1) = S(N)
270 :      S(N) = TEMP
280 :      COUNT = COUNT + 1
290 : NEXT N
300 UNTIL COUNT = 0
400 REM PRINT RESULTS
410 CLS: PRINT
420 FOR P = 1 TO 10
430 : PRINT NUM(P),, S(P)
440 NEXT P
```

The remarks are used to indicate the functions of the different sections of the program, which works like this.

Line 20 dimensions the two arrays used in the program. (This is not strictly necessary for ten numbers, but it would be if any more were to be sorted.)

Lines 40 to 70 input ten numbers from the keyboard and store them in the array NUM(10), using a **FOR ... NEXT** loop.

Lines 110 to 130 copy the numbers from NUM(10) to a second array S(10) which will be sorted, while NUM retains the numbers in the order in which they were typed in. Again, a **FOR ... NEXT** loop is used.

The lines from 210 to 300 sort the numbers in the array S(10) into ascending order. Two nested loops are used. The inner loop, a **FOR ... NEXT** loop, compares each element of the array S with the next, and swaps them over if they are in the wrong order. The outer loop - **REPEAT ... UNTIL** -

repeats until no swaps are made within the inner loop, which means the sorting is complete.

Lines 410 to 440 complete the program by printing the two sets of numbers.

SUBROUTINES

A subroutine is a section of program which is executed at a number of different times in the course of a program, but is written only once. The computer is diverted from the main program to the subroutine, and returns to the main program at the point from which it left.

The command **GOSUB** followed by a line number or a variable diverts the computer to the subroutine in much the same way as the **GOTO** command. The difference is that the end of a subroutine is marked by a **RETURN** command which sends the computer back to the instruction after the **GOSUB** command. As an example of two very simple subroutines type in the following program.

```
10  GOSUB 110      'INPUT A AND B
20  C = A+B
30  GOSUB 210      'DISPLAY RESULT
40  END

100 REM SUBROUTINE TO INPUT A & B
110 INPUT "A, B"; A, B
120 RETURN

200 REM SUBROUTINE TO DISPLAY
    RESULT
210 CLS
220 PRINT
230 PRINT A "+" B " = " C
```

240 RETURN

Lines 10 to 40 are the main program. Line 10 calls the subroutine at line 110, line 20 adds A and B and stores the result as C, and line 30 calls the subroutine at line 210. Line 40 marks the end of the program; as the last line to be executed is not the last line of the program we have to tell the ORIC not to go on to the next lines, which contain the subroutines.

The subroutine at line 110 inputs two numbers and stores them as A and B. The subroutine at line 210 clears the screen and prints the two numbers and their sum.

When the program is **RUN**, the computer begins, as always, at the lowest numbered line, line 10. This line calls the first subroutine, and the computer is diverted to line 110. Line 120 returns the computer to the instruction after the **GOSUB**, which is in line 20. The program proceeds as normal to line 30, which causes another diversion to line 210. Line 240 returns the computer to line 40, and the program ends.

So, a **GOSUB** command causes a diversion from the sequence of a program to a subprogram, and the **RETURN** command ends the diversion.

The use of subroutines saves a lot of effort in writing programs, as the program lines in the subroutine need to be written only once, instead of being retyped at every point in the program where they are needed. There is no limit on the number of times a subroutine may be called.

Another advantage of subroutines is that they can make the design of a program simpler. If you use subroutines for the repetitive and less important parts of a program the main program becomes

much easier for you, the programmer, to follow, and is therefore much easier to write.

Subroutines, like loops, may be nested - that is, the subroutines may call other subroutines, which may in turn call others. The maximum number of subroutines which may be nested like this is 24. However a subroutine may not be called by another subroutine which has been called by the first one, or an endless loop occurs, and the program will crash when it runs out of memory to store all the line numbers for the **RETURNS**.

STOPPING AND STARTING

In the last example program the new command **END** was used to indicate the last line of the program to be executed. The command tells the ORIC to stop running the program. There is a second command, **STOP**, which also stops programs, but the two have slightly different effects. **END** stops the program running, and the 'Ready' message is displayed, but when **STOP** is used the program halts and the message 'BREAK IN LINE xxx' is printed (xxx is the line number of the stop instruction). Unless you need to know where the program stopped, **END** is the one to use.

If a program has been stopped by one of these instructions, or by holding down CTRL and pressing C, it may be restarted with the instruction **CONT**. The program will continue from the instruction after the last one to have been executed. **CONT** may only be used immediately after the program has been stopped: if you have altered the program listing the message '?CAN'T CONTINUE ERROR' will be displayed. You may not use **CONT** within programs, as this will give a '?SYNTAX ERROR'.

ON ... GOSUB (GOTO)

Both **GOSUB** and **GOTO** may be used in a second type of decision command which selects one of a number of destinations depending on the value of a chosen variable.

```
ON N GOTO 100, 200, 300
```

results in

```
GOTO 100    if N = 1
GOTO 200    if N = 2
GOTO 300    if N = 3
```

Similarly,

```
ON P GOSUB 500,200,500,400,100
```

selects the P'th destination in the list and calls it as a subroutine. If the value of the variable is greater than the number of destinations in the list, or if it is zero or less, no destination is selected and the program continues at the next instruction.

The line numbers may be represented by variables, as they can with **GOTO** and **GOSUB**.

SUMMARY

LOOPS

There are two types of loop in BASIC:

- 1 **REPEAT ... UNTIL (condition)** in which the instructions between the **REPEAT** and **UNTIL** commands are repeated until the condition given after **UNTIL** is satisfied; and

- 2 **FOR V = A TO B STEP C ... NEXT** in which the instructions between the **FOR** and **NEXT** commands are repeated once for each value of V indicated by the **FOR ... STEP** command. A, B and C may be variables or numbers; V must be a variable. Loops may be nested one within another.

DECISIONS

IF (condition) THEN (instruction) ELSE (instruction) decides between two actions. The condition after **IF** is usually a test of a variable. There may be more than one instruction after **THEN** and **ELSE**, in which case colons (:) must be used to separate them. The complete **IF ... THEN ... ELSE** command must be in one line.

GOTO need not be included if it would be the only command after **THEN** or **ELSE**, but it must be if there is another command before it.

```
100 IF ACE = 1 THEN 20 ELSE 500
```

```
but 100 IF ACE = 2 THEN KING =  
7:GOTO20:ELSE QUEEN = 5:GOTO 40
```

SUBROUTINES

Diversions from a program, using **GOSUB** and **RETURN**. **GOSUB** calls the subroutine, **RETURN** at the end of the subroutine returns the ORIC to the instruction after the **GOSUB**.

ON ... GOSUB (GOTO)

Selects a destination from a list.

```
ON N GOTO A, B, C
```

causes a jump to the N'th destination, if there is an N'th item in the list.

ON P GOSUB X, Y, Z

causes a **GOSUB** to the P'th item in the list.

END halts the running of a program.

STOP stops a program and prints 'BREAK IN LINE xxx'.

CONT will restart a program if the program has not been altered.

CHAPTER 6

DATA AND PROGRAMS

So far we have not examined the ways in which information can be given to programs, and by which programs can print out information on to the screen. There are several ways of getting information into programs which we will examine in this chapter, and we will also examine in more detail the use of **PRINT** and other commands to present data on the display.

There are two commands, **GET** and **KEY\$**, which read characters directly from the keyboard. The two commands are similar, but there are important differences.

GET

The **GET** command halts a program and waits for a key to be pressed. The key character is stored in a specified variable.

```
10    PRINT "PRESS A KEY"  
20    GET A$  
30    PRINT A$  
40    GOTO 10
```

This program demonstrates how **GET** works. The **GET** command in line 20 causes the program to wait for a key to be pressed, then stores the key character in **A\$**. The program loops endlessly, **GET**ting and **PRINT**ing characters. (You may find that you can't stop the program in the usual way by pressing **CTRL** and **C**. If so, turn the **ORIC** over and use a pen or pencil to press the **RESET**

button which is behind the rectangular hole on the bottom of the machine.)

If you change the variable in line 20 from a string variable to a numeric variable, then only the number keys, and the keys used in relation to numbers (. + - and #) may be pressed. Pressing any other key will cause the ORIC to stop running the program and print a 'SYNTAX ERROR' message.

KEY\$

The function **KEY\$** gives a similar result to **GET**, but the operation of the **KEY\$** instruction is different. The program does not wait for the key-stroke, but gives the value of the last key to have been pressed. This is because whenever you press a key, the key code is stored in a 'buffer' (a special area of memory reserved by the ORIC for a specific purpose) in the memory. **KEY\$** reads the character in the buffer, and clears the buffer so that a subsequent **KEY\$** command won't read the same character again.

The **KEY\$** function is used like this:

```
10   A$ = KEY$  
20   PRINT A$  
30   GOTO 10
```

You will see that the program keeps on looping and printing even when no key has been pressed, and that each character you type is printed only once - the rest of the time the program prints blanks.

The **KEY\$** command can only be used with string variables.

The **GET** and **KEY\$** commands are useful in programs which ask the operator to select options, or answer Yes or No.

```
10  PRINT "PRESS Z FOR ZAP"
20  GET YN$
30  IF YN$ = "Z" THEN ZAP ELSE
    EXPLODE
40  GOTO 10
```

READ AND DATA

To enter large amounts of data which will be the same each time the program is run we use the last type of data entry command: **READ**. Look at this program:

```
10  DIM MTH$(12):  DIM DAYS(12)
20  FOR M = 1 TO 12
30  :  READ MTH$(M)
40  :  READ DAYS(M)
50  :  PRINT MTH$(M)"    HAS
    "DAYS(M)"  DAYS"
60  NEXT

1000 DATA  JANUARY, 31, FEBRUARY,
      28
1010 DATA  MARCH, 31, APRIL, 30
1020 DATA  MAY, 31, JUNE, 30
1030 DATA  JULY, 31, AUGUST, 31
1040 DATA  SEPTEMBER, 30, OCTOBER,
      31
1050 DATA  NOVEMBER, 30, DECEMBER,
      31
```

The information for this program is written into the program in lines 1000 to 1050. Each item of data is separated from the next by a comma (or the end of a line). The information is copied into the arrays **MTH\$** and **DAYS** by the **READ** commands

in line 30 and 40.

This method of entering information is much easier to use than simply writing it into the program by setting variables. Imagine the typing involved in entering lots of lines like

```
MTH$( 2 ) = "FEBRUARY "
```

With **DATA** lines you can lay out the information in a clear tabular form and check it much more easily. You can also alter it if you need to without touching the main program.

When a program is running, the ORIC uses a 'pointer' stored in its memory to keep track of how many data items it has read. Every time a **READ** command is met, the data item indicated by the pointer is copied into the variable, and the pointer moves on to the next item. This means you must have a **DATA** entry corresponding to each occurrence of **READ**: the program will stop if there are too many **READ** commands, and the ORIC will print 'OUT OF DATA ERROR'. The command **RESTORE** can be used to restore the data pointer to the first data item in the program.

The variables used in the **READ** command must match the type of data in the corresponding **DATA** entry. A **READ** with a string variable will read anything as a string, but **READ** with a number variable must find a number in the **DATA** entry or a 'SYNTAX ERROR' will result. Integer variables must find integral numbers; real variables may read decimal fractions as well.

MORE ABOUT PRINTING

So far, we have described the use of **PRINT** (or ?) simply to display single items of data on the screen.

The ORIC has a number of extra facilities which allow you to control the screen layout and produce clear and orderly output from programs.

PUNCTUATING PRINT STATEMENTS

You can **PRINT** more than one item on a single line by including 'punctuation' in the **PRINT** command. If you use commas to separate the items, they will be printed on the same line, with three spaces between each item. For example,

```
PRINT "A", "B", "C", "D", "E", "F"
```

will give a display:

```
A      B      C      D      E      F
```

The characters are all on the same line and equally spaced.

The semicolon (;) can be used in a similar fashion. This leaves no space between successive items in the **PRINT** statement.

```
PRINT "A"; "B"; "C"; "D"; "E"; "F"
```

will display

```
ABCDEF
```

Printing numbers, real and integer variables, the same rules apply, except that an extra space is always printed after each number. This means that using commas puts four spaces between numbers, and using semicolons will result in one space between the numbers.

The effect of commas and semicolons is not confined to the **PRINT** statement in which they appear. If you end one **PRINT** statement with a comma or

semicolon then the data printed by the next **PRINT** command will appear on the same line, with three or no spaces as appropriate.

You can exercise even more control over the formatting of screen displays using two more BASIC commands - **SPC** and **TAB**.

SPC(N)

SPC tells the ORIC to **PRINT** N spaces.

```
CLS: PRINT "I AM" SPC(10) "ORIC"
```

will print 10 spaces between 'I AM' and 'ORIC'.

TAB(N)

The **TAB** command instructs the ORIC to move the cursor N spaces from the left side of the screen before printing. For example,

```
PRINT TAB(17) "ORIC"
```

prints 'ORIC' in the middle of a screen line.

NOTE On many ORICs the **TAB** command does not work properly. If you add 14 to the number N you will get something like the right effect, but you will find that the **TAB** command will then have exactly the same effect as the **SPC** command and will print spaces as well as moving the cursor to the point where it will start printing.

There is a function **POS** corresponding to the **TAB** command which returns the number of the column in which the cursor is placed. For example:

```
PRINT SPC(10);: PRINT POS (0)
```

gives the value 12, since the cursor was in column 11 after the first **PRINT** command. (The number in brackets has no significance - any number will do.)

CURSOR CONTROL

Just as we can use the cursor keys to move the cursor round the screen, so we can use a program to move it. The ORIC interprets four characters as control codes governing the cursor movement. These characters are shown in the table (and explained in more detail later).

CODE	MOVE CURSOR
8	LEFT
9	RIGHT
10	DOWN
11	UP

Cursor Control Codes

The characters can be printed using the **CHR\$** function. For example:

```
10  CLS
20  PRINT "TOP LINE"
30  PRINT CHR$(10) CHR$(10)
    CHR$(10) CHR$(10)
40  PRINT "NEXT LINE"
```

prints the two messages with blank lines between them over which the cursor has moved.

Turning off the Cursor

In some applications, such as drawing pictures or patterns on the screen, you may like to turn off the flashing cursor. To do this, press CTRL and Q together. To turn the cursor back on, repeat the procedure.

You can do this in a program with the statement:

```
PRINT CHR$(17)
```

Again, repeating the statement will turn the cursor back on.

A further command which controls the cursor is

```
PRINT CHR$(30)
```

which moves the cursor to the top left corner of the screen, known as the 'home' position.

There are more details of the use of **CHR\$()** to control the display in Appendix 5.

The way in which data is presented on the screen is an important part of the program and can make the difference between a program being easy to use, or frustrating and confusing.

PLOTTING

There is a second instruction which is used for writing on to the screen - the **PLOT** instruction. Clear the screen and try:

```
PLOT 12, 15, "THE PLOT THICKENS"
```

The two numbers in the **PLOT** command are the coordinates of the position on the screen where the

message is to begin. The first number is the number of character spaces across the screen, the second is the number of lines down from the top. The example begins writing in the twelfth character space from the left of the screen and fifteen lines down from the top.

SUMMARY

GET is used to copy keystrokes from the keyboard into variables. The program waits for a key to be pressed before continuing.

KEY\$ is similar but the program does not pause.

READ and **DATA** are used to copy large amounts of information into variables without writing lots of program lines.

PRINTING

There are a number of special commands which allow control of the format of data displayed on the screen.

PLOT is used to position text accurately on the screen.

CHAPTER 7

PIECES OF STRINGS

String variables are used to store sequences - 'strings' - of characters. There are a number of BASIC commands which are used to manipulate strings, and these are described in this chapter.

We saw in Chapter 4 that a string variable can store a sequence of characters. For example:

```
NAME$    = "WILLIAM SHAKESPEARE"  
GAME$    = "SPACE INVADERS"  
REFERENCE$ = "ABC123D"
```

A string can hold up to 255 characters. These may be letters, figures, punctuation marks, spaces; in fact any of the characters the ORIC can print. The number of characters in a string can be counted using the function LEN. Try this:

```
10  NAME$ = "JOHN SMITH"  
20  PRINT LEN(NAME$)
```

This program prints the number 10, which is the number of characters in NAME\$ (including the space).

TYING STRINGS TOGETHER

Strings can be added together.

```
10  A$ = "FLORENCE"  
20  B$ = "NIGHTINGALE"  
30  SPACE$ = " "  
40  NAME$ = A$ + SPACE$ + B$
```

50 PRINT NAME

Notice that strings don't add like numbers. B\$ + A\$ does not give the same result as A\$ + B\$. Don't forget that you can't have a string longer than 255 characters. If a program tries to add too many characters together, the ORIC will print a 'STRING TOO LONG ERROR' message on the screen and stop the program.

CUTTING STRINGS

There are three BASIC functions which are used to separate pieces from strings. The functions are:

LEFT\$
RIGHT\$
and **MID\$**

LEFT\$ and **RIGHT\$** are both very similar in operation. **LEFT\$** is used to read characters from the beginning of a string; **RIGHT\$** reads from the end of the string. The functions are used like this

LEFT\$ (String, Number)

The string in the brackets may be a variable or an actual sequence of characters between quotation marks ("), and the number may be a number or a variable.

For example:

```
PRINT LEFT$ ( "ABCDEFGH" ,4)
```

prints 'ABCD', and

```
10 S$ = "UVWXYZ"  
20 N=3  
30 R$ = RIGHT$ (S$,N)
```

40 PRINT R\$

prints 'XYZ'.

The third function, **MID\$**, is a little more complex, as we have to specify not only the string and the number of characters to be read, but also where in the string the characters are to be found.

```
PRINT MID$ ("12345678", 3, 4)
```

prints four characters, beginning at the third, of '12345678': so it prints '3456'. If you do not specify the number of characters to be read, all characters to the right of the specified character are included. So

```
PRINT MID$ ("ABCDEFGHIJK", 4)
```

prints 'DEFGHIJK'.

In Chapter 10, Making a Noise, you will discover that strings are useful for storing tunes in a compact manner, and **MID\$** is used to extract one note at a time from the strings.

NUMBERS AND LETTERS

Computers cannot handle characters (numbers, signs and letters) directly - they use numbers to symbolise the characters. Each letter, and each of the other characters the ORIC can print, is represented by a different code number. There are several systems used in different computers for deciding which code number represents which letter. The ORIC uses a system called **ASCII** - the American Standard Code for Information Interchange - which is the most common system in micros. The table in Appendix 4 shows how the ASCII code relates letters and numbers.

From this table we see that the ORIC uses a code number 65 to represent the letter A, code number 66 for B and so on. Even the numeric characters (digits) have codes. For example the character '9' has the code number 57. What use is this knowledge? There are two functions in BASIC which allow us to convert characters to numbers and numbers to characters.

The function **CHR\$** converts a number to a string containing the corresponding character.

```
10   C = 65
20   A$ = CHR$(C)
30   PRINT A$
```

prints a letter 'A'. You can use a variable with **CHR\$**, as in the example, or a number.

```
PRINT CHR$(42)
```

prints a *.

CHR\$ can only give one character at a time, and the number or the variable inside the brackets must be less than 255, or the ORIC will complain of an 'ILLEGAL QUANTITY ERROR'.

The function **ASC** complements **CHR\$**. **ASC** finds the ASCII codes of characters.

```
10   B = ASC("B")
20   PRINT B
```

prints 66, which is the ASCII code for the letter B. You can use string variables with **ASC**:

```
10   B$ = "B"
20   PRINT ASC(B$)
```

also prints 66. If the variable contains more than one character, **ASC** gives the code for the first character only; so

```
PRINT ASC("ABCDE")
```

prints 65.

FIGURES IN STRINGS

A string can contain any characters, including the number characters. There are two functions which can be used to make strings of number characters from actual numbers and to find the numeric value of the number characters in a string.

The function **STR\$** creates from a number a string containing the characters of that number.

```
10  A$ = STR$(12345)
20  PRINT A$
```

converts the single number 12345 into a string by putting the characters '12345' into A\$ and prints the string.

The complementary function to **STR\$** is **VAL**. **VAL** evaluates the numerical characters in a string.

```
10  A$ = "123456"
20  A = VAL(A$)
30  PRINT A
```

prints the number 123456.

If the string evaluated contains letters as well as numbers then only the numbers which appear to the left of all the letters are counted by **VAL**. This means that

```
PRINT VAL("123ABC45")
```

prints 123.

The signs + and - may appear before the number characters. They will be treated by VAL as the sign of the number. If there is a # at the beginning of the string the VAL function will consider not only the characters from 0 to 9 but also the letters from A to F. The string will be considered as a hexadecimal number (if you are unsure about hexadecimal numbers they are explained in Appendix 8).

TESTING STRINGS

Strings can be compared with each other, as numbers can, after the IF and UNTIL commands.

```
10 INPUT "WHAT'S THE PASSWORD";  
P$  
20 IF P$ <> "LEMON CURRY" THEN  
PRINT "WRONG !!!":EXPLODE
```

A routine like this one can be used to check that the input is suitable for the program. Another use for comparing strings is when using the GET command in conjunction with a 'menu'. (In computing, a menu is a list of options displayed on the screen to the program operator.) The next routine displays a simple menu, GETs the selection from the keyboard and then calls the selected routine.

```
10 CLS  
20 PRINT: PRINT "SELECT OPTION:"  
30 PRINT: PRINT "P ... PING"  
40 PRINT: PRINT "Z ... ZAP"  
50 PRINT: PRINT "E ... EXPLODE"
```

```
60    PRINT: PRINT "ESC . STOP
      PROGRAM"
70    GET K$
80    IF K$ = "P" THEN 1000
90    IF K$ = "Z" THEN 2000
100   IF K$ = "E" THEN 3000
110   IF K$ = CHR$(27) THEN END
120   GOTO 70
1000  PING: GOTO 70
2000  ZAP: GOTO 70
3000  EXPLODE: GOTO 70
```

WARNING

Be careful when using **STR\$** and making comparisons. If you compare "123" with **STR\$(123)** you will find that the ORIC thinks they are not the same. This is because for some reason an extra invisible character is added to the beginning of the string. The character is a **CHR\$(2)**, which cannot be seen when the string is printed but which may set the colour of the ink to green on the line on which the string is printed. We have no idea why the ORIC does this: you can get round the problem by using **MID\$** to remove the first character of the string.

```
A$ = MID$(STR$(123),2)
```

As well as testing to see if two strings are the same, you can use the 'greater than' and 'less than' (> and <) tests to compare strings.

```
10    IF "B" > "A" THEN PING
```

This works because the letters are stored as numbers. The ORIC is in fact comparing the ASCII codes of the letters in the strings. This is very useful because it allows strings to be sorted into alphabetical order:


```
10   CLS
20   DIM W$(20)
30   COUNT = 0
40   REPEAT
50   :   COUNT = COUNT + 1
60   :   INPUT "ENTER A WORD";
       W$(COUNT)
70   UNTIL W$(COUNT) = "ZZZ" OR
       COUNT = 20
80   IF W$(COUNT) = "ZZZ" THEN
       COUNT = COUNT - 1
90   CLS: GOSUB 1000   'PRINT
       UNSORTED LIST

100  REM   SORT STRINGS
110  REPEAT
120  :   SWAPS = 0
130  :   FOR I = 1 TO COUNT
140  :       IF W$(I+1) >= W$(I) THEN
190      'COMPARE 2 STRINGS
150  :       TEMP$ = W$(I+1)   'SWAP
       STRINGS
160  :       W$(I+1) = W$(I)
170  :       W$(I) = TEMP$
180  :       SWAPS = SWAPS + 1
       'COUNT SWAPS
190  :   NEXT I
200  UNTIL SWAPS = 0
210  GOSUB 1000   'PRINT SORTED
       LIST
250  END

1000 REM SUBROUTINE TO PRINT
       RESULTS
1010 PRINT
1020 FOR P = 1 TO COUNT
1030 :   PRINT W$(P)
1040 NEXT P
1050 RETURN
```

This program works in the same way as the sorting program at the end of Chapter 5. You can input up to twenty strings, which may contain more than one word. If you wish to sort fewer than 20 strings, then enter ZZZ as the last string; this will tell the program that there are no more strings to come.

If you run this program a few times you will discover how effective string sorting can be. Notice that any characters will be sorted into ASCII order, so strings like */@£?% and /@\$&)# will be sorted. You will find that lower case letters are treated as completely different, because of course they are represented by different codes, and the ORIC is interested only in the codes. The lower case letters have higher codes than the capitals, so they appear at the end of the sorted list. (You can switch in and out of lower case by holding down CTRL and pressing T).

SUMMARY

A string is a sequence of characters. These can be stored in string variables, which are distinguished by \$ after the variable name.

LEN(string) gives the number of characters in a string.

Strings can be added together:

```
PRINT "ABC" + "DEF" + "GHI"
```

prints ABCDEFGHI.

LEFT\$, RIGHT\$ and MID\$ are used to separate parts of strings.

LEFT\$(string,N) takes the first N characters of the string.

RIGHT\$(string,M) reads the last M characters.

MID\$(string,P,Q) reads Q characters, starting at the P'th.

ASCII code is used by the ORIC to symbolise characters. There is a table of the ASCII codes for the characters in Appendix 4.

CHR\$ converts a number to the equivalent character

```
PRINT CHR$(65)
```

prints A.

ASC gives the ASCII code for a character

```
PRINT ASC("A")
```

prints 65.

STR\$ turns a number into a string of number characters

```
10  A$ = STR$(123)
20  PRINT A$
```

VAL evaluates the numerical characters in a string

```
PRINT VAL("123ABC789")
```

prints 123.

COMPARISONS.

Strings can be compared using **IF** and **UNTIL** and the relational operators =, <, and >. This is useful for testing inputs and sorting strings.

CHAPTER 8

FUNCTIONS

A 'function' in computing is an instruction which performs a calculation on a number. There are a number of functions available on the ORIC. For example, in Chapter 7 several functions were described which operate on numbers to give strings, on strings to give other strings, or on strings to give numbers. In this chapter we will look at some more of the functions available on the ORIC.

SQUARE ROOTS

The square root of a number is calculated by the function **SQR()**. (The square root of a number is the number which when multiplied by itself, or *squared*, gives the first number.) Try

```
PRINT SQR(4)
```

The ORIC prints 2, because 2 squared ($2*2$) is 4.

```
10 FOR N = 1 TO 10
20 PRINT N, SQR(N)
30 NEXT
```

prints the numbers 1 to 10 and their square roots.

ABSOLUTE VALUES

The function **ABS** finds the **absolute** value of a number: the value of the numerical part, disregarding the sign. The function changes

negative numbers to positive numbers, but has no effect on positive numbers.

```
PRINT ABS( 123.456)
```

prints 123.456 - no change. But

```
PRINT ABS( -543.345)
```

prints 543.345 - the minus sign has been removed.

INTEGER CONVERSION

The function **INT** removes the fractional part of a number and returns the next lower whole number. Try

```
PRINT INT(123.4567)
```

Only the whole number part - 123 - is printed.

Be careful with numbers less than zero. The **INT** function finds the first whole number lower than the number you give it. This means that for negative numbers the answer is not the whole part of the initial number, but one less (or minus one more!). So,

```
PRINT INT(-2.875)
```

prints -3.

SIGNS

SGN() returns a value which indicates the sign of a number. If the number is positive, the result is 1; if the number is zero, the result is zero; and if the number is less than zero, **SGN** returns -1.

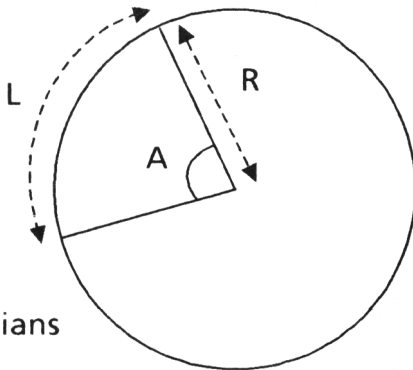
```
PRINT SGN (5)    prints 1
PRINT SGN (0)    prints 0, and
PRINT SGN (-7)   prints -1.
```

TRIGONOMETRY

The trigonometrical functions of sine, cosine, tangent and arctangent are available in ORIC BASIC. They are written:

```
SIN()
COS()
TAN()
ATN()
```

Note that the angles on which these functions operate must be given in **radians**, not in degrees. A radian is the ratio of the length of an arc of a circle to the radius. In the diagram, the angle A in radians is L/R . The circumference of a circle is 2π times its radius ($2\pi R$), so it follows that 360 degrees is equivalent to 2π radians.



The angle A is L/R radians

From this we can work out that 1 degree is $2\pi/360$ radians (about 0.175), and 1 radian is $360/2\pi$ degrees (about 57.3°). To convert from degrees to radians multiply by $2\pi/360$. This is easy on the ORIC, as the value of π is held in a preset variable called PI. To convert from radians to degrees multiply by $360/2\pi$.

So, the sine of 45 degrees (0.7071) is given by:

```
10 RAD = 2*PI/360
20 PRINT SIN(45*RAD)
```

The arctangent of 1 (45 degrees) is given by:

```
10 RAD = 2*PI/360
20 ANGLE = ATN(1)/RAD
30 PRINT ANGLE
```

LOGARITHMS

Both natural logarithms and base 10 logarithms can be used with ORIC BASIC. Natural logs are provided by the function LN().

```
PRINT LN(10)      prints 2.30258509
PRINT LN(5000)    prints 8.5171932.
```

The antilogs of natural logarithms are calculated by EXP(). Try

```
PRINT EXP(LN(100))
```

The answer is 100.

Logs can be used to calculate roots. This program finds the cube root of 8.

```
10 A = 8
20 R = LN(A)/3
```

```
30 PRINT EXP(R)
```

The answer is 2, because 2 cubed ($2*2*2$) is 8.

Base 10 logarithms are used in a similar way. There is no antilog function, as antilog can be calculated easily, using powers of 10. Try

```
PRINT 10↑LOG(55)
```

The result is 55. In general, 10^X gives the antilog of X. Logs to base 10 can be used in a similar fashion to natural logs. This program calculates fifth roots.

```
10 N = 243
20 R = LOG(N)/5
30 PRINT 10↑R
```

The program will print 3, because 3 to the power 5 (3^5 or $3*3*3*3*3$) is 243.

RANDOM NUMBERS

The ORIC has a function which provides random numbers. The function is **RND()**, which prints a number between 0 and 1. Try **PRINTing RND(1)** a few times. You will get a different number each time.

The random numbers provided by **RND** are not truly 'random' - it is very difficult and expensive to make a machine which will give perfectly random numbers. If you switch your ORIC off and on, and immediately type **PRINT RND(1)**, you will get the same number every time (on our ORIC it's .270011996). The random numbers are calculated from a starting number, or 'seed', and any seed will always produce the same sequence of numbers.

The random number calculation can be controlled by altering the number in the brackets.

You can give the ORIC a new seed for its random number generator by putting a negative number in the function. Try `PRINT RND(-1)`. The resulting number (probably about `3E-08`) is not much use, but the effect of the command is to give a new seed number to the random number generator. Try `PRINT`ing five numbers with `RND(1)` and make a note of them. If you then re-seed the random number generator with 1 by `PRINT`ing `RND(-1)` again, and re`PRINT` five numbers with `RND(1)`, you will get the same sequence as before. Each negative number sets off a different sequence.

There is a way of obtaining a more nearly random sequence of 'random' numbers. If the seed given to the random number generator is different each time, the sequences will be different. We can get an unknown, varying seed by using the contents of memory locations 630 and 631, which hold a number between 0 and 65535 which is decremented by the ORIC every hundredth of a second. This means that

```
PRINT RND(-DEEK(630))
```

will give an unknown seed to the random number generator. (`DEEK` is explained in Chapter 15, Advanced Programming.)

To sum up, if the number given to the `RND` function is greater than zero, a number from a sequence of 'random' numbers is returned by the function. If the number given to the function is negative, a new sequence of random numbers is produced.

Finally, if the number given to **RND** is zero, the number returned is the same as the one before.

Dice Throwing

We can use random numbers in programs to imitate the throwing of dice. Try this:

```
10  A = RND(-DEEK(630))
20  CLS: PRINT: PRINT "PRESS ANY
    KEY TO THROW THE DICE"
30  REPEAT
40  GET K$
50  FOR I = 1 TO 10
60  DICE = INT( (RND(1)*6) + 1)
70  D$ = "YOUR NUMBER IS " +
    MID$(STR$(DICE),2)
80  PLOT 5,10, D$
90  WAIT 5: NEXT
100 UNTIL K$ = CHR$(27)
```

This program will throw the die every time you press a key. Pressing the **ESC** key will stop the program. The use of **MID\$** in line 70 is to remove the superfluous character (**CHR\$(2)**) from the beginning of the string created by **STR\$**. (Refer to Chapter 7 for an explanation of this.)

DEFINING YOUR OWN FUNCTIONS

The ORIC allows you to define up to twenty six functions of your own, which can be used throughout a program. Functions are defined by the command **DEF FN**. They can then be used like any other functions in calculations and tests.

```
10  DEF FNA(N) = PI * N↑2
20  PRINT "PROGRAM TO CALCULATE
    AREAS OF CIRCLES"
30  PRINT
```

```
40    INPUT "ENTER THE RADIUS";R
50    PRINT: PRINT "AREA OF A
      CIRCLE RADIUS "R" IS "FNA(R)
60    IF FNA(R) < 100000 THEN 30
```

This program uses a function FNA which is defined in line 10. The variable N in the definition is a 'dummy' variable - any variable name can be used when the function is used later in the program. The function is used in lines 50 and 60.

You can define up to twenty six functions, with names from FNA to FNZ.

SUMMARY

There are several built-in functions in ORIC BASIC which operate on numbers or variables.

SQR(N) calculates the square root of a number N.

ABS(N) returns the absolute value of a number - changing negative numbers into positive ones, and leaving positive numbers unchanged.

INT(N) returns the integer value of N, removing any fractional part.

SGN(N) gives 1 if N is positive, 0 if N is zero and -1 if N is negative.

The trigonometric functions

SIN(N)
COS(N)
TAN(N)
ATN(N)

return the values for the angle N (which must be in radians).

LOG(N) returns the base 10 logarithm of a number, while **LN(N)** returns the natural logarithm.

RND(1) returns a pseudo random number between 0 and 1.

DEF FN allows you to define your own functions within programs.

CHAPTER 9

LOGICAL THINKING

As well as doing arithmetic, the ORIC can perform tests to compare numbers and strings. We have already seen this when using **IF...THEN** and **UNTIL...** In this chapter we will examine in more detail the way in which the ORIC makes comparisons.

Consider the program line

```
220 IF A = 3 THEN 500
```

The line instructs the ORIC to branch to line 500 if the variable A holds the value 3. What does the ORIC do when it encounters this program line?

The first thing the ORIC must do is decide whether A is equal to 3; or to put it another way, whether 'A = 3' is true. The ORIC tests this, and if the expression is **TRUE**, it returns an answer of -1; if the expression is **FALSE**, the answer is 0. If the result is true (-1), the commands after **THEN** are obeyed; if the answer is false (0) the commands after **ELSE** (if there are any) or the instructions on the next program line will be obeyed.

Why are we telling you all this? Because the ORIC can compare numbers and return true or false values without **IF** or **UNTIL**. Try this short program:

```
10 A = 3
20 PRINT A = 3
```

The program prints -1. If you change line 10 to

```
10  A = 5      (or any other number)
```

the program will print 0.

This applies to all the other comparisons listed in Chapter 5 as conditions for **UNTIL**:

=	Equal
<	Less than
>	Greater than
<>	Not equal
>=	Greater than or equal
<=	Less than or equal

Try this:

```
10  A = 5
20  PRINT A <= 7
```

If you try out the other tests, you will find they all behave in the same way.

LOGIC

If two tests are combined, the same true and false answers are still obtained.

```
10  A = 5
20  PRINT ( A<7 ) AND ( A>3 )
```

Now, if the ORIC evaluates simple relational expressions such as $A < 7$, $A > 3$ as -1 or 0, what happens when two are combined, and what does the **AND** do?

There are three BASIC commands which can be used with relational expressions: **AND**, **OR** and **NOT**. Forgetting about numerical representations of true and false for the moment, let's look at what these commands do.

Using **AND** to relate two expressions, as in the example above, it seems fairly obvious that the final result will be true only if both smaller expressions are true. This is in fact what happens. Here is a table of the possible combinations, with the two simple expressions represented by X and Y. (This type of table is called a 'truth table'.)

X	Y	X AND Y
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Truth Table for **AND**

The command **OR** also does the obvious thing, returning TRUE if X or Y or both are true.

X	Y	X OR Y
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Truth Table for **OR**

NOT changes from true to false, and vice versa, so

```
10  A = 3
20  PRINT NOT A>10
```

prints -1 (**TRUE**). Here is the truth table for **NOT**.

X	NOT X
TRUE	FALSE
FALSE	TRUE

Truth Table for **NOT**

All this is fairly easy to follow. But the ORIC is using numbers to represent **TRUE** and **FALSE**. How does it work out the answers?

To understand this, you must understand how the ORIC stores numbers in binary notation (Appendix 8 contains an explanation). **AND**, **OR** and **NOT** are actually applied by the ORIC to the binary representations of the numbers. The truth tables for the three operators remain the same, but true and false are replaced by 1 and 0. The truth table for **AND** therefore changes to the form shown on page 82.

Each number contains more than one binary digit (or 'bit'). The ORIC applies **AND**, **OR** and **NOT** to the numbers by comparing bits at the same place in each number. Suppose A and B were two 4-bit numbers:

A = 1 1 0 0 B = 1 0 0 1

The digits for (A **AND** B) are found by comparing the corresponding digits of A and B. So, bit 1 of A

A	B	A AND B
1	1	1
1	0	0
0	1	0
0	0	0

Numerical Truth Table for **AND**

AND B (counting from the right) is (0 **AND** 1), which is 0. Bit 2 is 0 **AND** 0 which is also 0. Proceeding thus, we find

$$A \text{ AND } B = 1000$$

By similar means we can show that if P is 1010 and Q is 0011, P **OR** Q is 1011; and if Z is 0101, **NOT** Z is 1010.

Returning to the ORIC's comparisons of **TRUE** and **FALSE**, we can now see why the numbers 0 and -1 are used. The ORIC uses the **two's complement** system to represent negative numbers. This means that -1 is stored as 1111, while 0 is stored as 0000. (The ORIC actually uses 16 bits to represent each, but it's easier to follow with only 4.) This means that **TRUE** is a number which is **all ones**, and **FALSE** is **all zeros**, so the **AND**, **OR** and **NOT** commands can work on the individual bits and produce the right answer.

It may seem that all this knowledge is of little use when programming in BASIC, but it can be used to simplify some programs which involve tests.

```
320 IF (A+B)>6 THEN Z=7 ELSE Z=0
```

could be replaced by

```
320  Z = -((A+B)>6)*7
```

and

```
500  IF P=Q THEN T=5 ELSE T=3
```

is equivalent to

```
500  T = 3-2*(P=Q)
```

AND can be used to keep numbers within certain limits by **masking**. Think about what would happen if any binary number was **ANDed** with the number 00001111. The four highest bits of the result (the left four) would be zero, whatever the other number, because 0 **AND** X is always 0. The four lower bits would be the same as those of the other number. For example

```
10101010 AND 00001111 = 00001010
```

We can therefore use **AND** to copy only a part of a number. Look at this example:

```
230  A = A + 1
240  IF A > 15 THEN A = 0
```

15 in binary is 00001111. If A is increased to 16 (00010000) then A **AND** 15 will be 0. We could replace the two program lines with

```
230  A = (A+1) AND 15
```

These uses of relations and logical operators may seem more confusing than using **IF** to perform tests, but they are also faster, which means that in a long program, or one which will loop many times, considerable time savings can be made.

SUMMARY

The ORIC can not only handle numerical calculations: it can solve logical problems too. Relational tests such as those performed after IF are represented numerically.

TRUE is represented by the number -1. There is a preset variable called TRUE with this value.

FALSE is represented by the number 0. There is a preset variable called FALSE which contains zero.

There are three logical operators.

AND A AND B is TRUE if both A and B are TRUE.

OR A OR B is TRUE if either A is TRUE or B is TRUE or both are TRUE.

NOT NOT A is TRUE if A is FALSE.

CHAPTER 10

MAKING A NOISE

The ORIC is capable of producing a wide variety of musical (and non-musical!) sounds, because it contains a special sound generator chip which is dedicated to this task.

It is this chip which produces a click each time you press a key, and the beep when you press **RETURN** or **CTRL**. These noises are to help you be sure that you have pressed the right key.

You can turn these sounds off by holding down **CTRL** and pressing **F**. The keyclick will remain turned off until you repeat the **CTRL F** sequence. The same effect can be achieved within a program by inserting a line such as

```
100 PRINT CHR$(6)
```

Again the keyclick will be turned off until another such line is encountered. Such a mechanism is known as a toggle, because every time it is encountered, the computer toggles between each of two possible states. This is not a very precise method of control, since in a complex program, it may be hard to predict which state exists at any particular time.

The most positive way to control the keyclick is to use the **POKE** command to alter one of the system variables. This is done by inserting a line

```
100 POKE #26A,11
```

which sets bit 4 to 1, in location #26A, and turns off the keyclick permanently until it is specifically restored by another line

```
200 POKE #26A,3
```

(The command **POKE** is explained in Chapter 15, Advanced Programming.)

When experimenting with the sound facilities of the ORIC, it is often useful to turn off the keyclick to prevent the noise made when pressing a key from interfering with other sounds being produced.

The ORIC has four BASIC commands to allow easy use of some of the sound facilities :

ZAP
EXPLODE
SHOOT
PING

These commands, mentioned in Chapter 3, cause the ORIC to make those four sounds. You may also have encountered the **PING** sound when trying to enter a program line longer than 76 characters. **PING** may also be heard by pressing CTRL and G, or by entering the command

```
PRINT CHR$(7)
```

These commands may be used within a program, for example,

```
10 FOR I = 1 to 10  
20 ZAP  
30 NEXT  
40 EXPLODE
```

will unleash a series of **ZAP**'s, leading to an explosion.

If line 20 is changed to incorporate a different sound, a delay must be inserted in the **FOR ... NEXT** loop, to allow the sound to finish before the next one begins. For example, change line 20 to

```
20 SHOOT
```

and add

```
25 WAIT 25
```

Line 25 controls the pause between shots, so it is easy to turn the rapid fire pistol into a machine-gun, by changing line 25 to

```
25 WAIT 10
```

If you use these commands within a program, remember that a **WAIT** command is necessary between successive **SHOOTs**, **EXPLODEs** and **PINGs**.

The ORIC is capable of more sophisticated sound synthesis than this, at the expense of a little more programming.

THE SOUND GENERATOR CHIP

The sound generator chip within the ORIC provides three tone channels, each of which can play musical notes spanning eight octaves. The tone channels can be played in any combination, so that chords and harmonies can be created. The volume of each channel is controllable, as is the 'shape' or envelope of the sound on each channel.

A white noise channel is provided, which can be mixed with any of the tone channels, in any combination.

The chip can be described as 'intelligent'. This means that data concerning the sound you wish to generate can be passed to the chip, and the computer can continue running your program as the sound is made. For this reason, using the sound generator commands will not slow the running of programs.

To give you control over these functions, the ORIC has three BASIC commands;

SOUND MUSIC PLAY

each of which is followed by several numbers, or parameters, indicating the channel or channels required, the note to be played, its volume, etc.

The least complicated of these commands is **SOUND**.

SOUND

The **SOUND** command must be followed by three numbers, separated by commas, which specify:

- 1 Which tone channel or channels are to be used, and whether or not white noise is to be added.
- 2 The Period of the tone.
- 3 The Volume of the tone.

A typical use of sound would be

```
SOUND 1,500,7
```

where channel 1 has been selected, and a note of period 500 played on that channel at volume 7.

Try this example in immediate mode - the tone will continue until you press a key (unless you've turned off the keyclick). Try varying the parameters to get different notes at different volumes, but don't change the channel number, as channel 1 is the only channel useable without extra programming, as we'll see in a moment.

SOUND Channel,Period,Volume

The three sound parameters work as follows (the numbers in brackets in the following paragraphs are the allowable range of values for these parameters - if you exceed them an error message is displayed).

Channel (1-6) determines which of the three tone channels is to be used: 1, 2 or 3 select these channels, whilst 4, 5 and 6 select channels 1, 2 and 3 respectively, but mix in the white noise channel. Unfortunately, this doesn't work on many ORICs, and only channels 2 and 3 are useable with white noise. The effect of the channel parameter is shown in this table:

CHANNEL PARAMETER	EFFECT
1	Channel 1 ON
2	Channel 2 ON
3	Channel 3 ON
4	Channel 1 ON + noise
5	Channel 2 ON + noise
6	Channel 3 ON + noise

NOTE You will only hear the effect of a **SOUND** command which uses channel 1, unless the **SOUND** command is

followed by a **PLAY** command. This is explained later in this chapter.

Period (0-65535) controls the pitch of the tone, and though any real number up to 65535 is allowed, there are only 4096 different periods. This means that the pitch obtained by the period command cycles every 4096, so that:

```
SOUND 1,100,7
SOUND 1,4196,7
and SOUND 1,61540,7
```

all sound identical.

Volume (0-15) controls the volume of the tone, with 15 as the loudest. If you use a volume of 0, control of the sound produced is passed to a subsequent **PLAY** command.

This program illustrates the full range of tones available using the **SOUND** command.

```
10  FOR I = 4096 TO 0 STEP -1
20  SOUND 1,I,7
30  NEXT I
```

Notice that the lowest tone is created with a period of 4096. If the period is greater than this, the cycle will start again. To hear this, replace line 10 with:

```
10  FOR I = 8192 TO 0 STEP -1
```

You will remember that it is not normally possible to use channels other than channel 1 with the **SOUND** command alone. To create chords and harmonies using the other channels you must use the **PLAY** command.

PLAY

The **PLAY** command is used to control the tone and noise channels and to define the 'shape' or envelope of the tones. The **PLAY** command must be followed by four numbers, separated by commas, which specify:

- 1 Which tone channels are to be switched on (any combination of the three channels may be specified).
- 2 To which of the channels white noise is to be added.
- 3 The envelope shape to be applied to the tone.
- 4 The 'period' of the envelope.

A typical use of **PLAY** would be to allow different tones on each of the three channels to be produced simultaneously. For example:

```
10  SOUND 1,300,7
20  SOUND 2,240,7
30  SOUND 3,200,7
40  PLAY 7,0,0,0
```

will play a chord, which will continue until you press a key (unless you've turned off the keyclick).

The **PLAY** command in line 40 selects all three channels to be played at once, with no noise added to any channel. The envelope mode is set to 0, indicating that no envelope is selected, and in this case the value of the envelope period is irrelevant.

The four **PLAY** parameters are Channel, Noise, Envelope and Envelope period. The command has the format:

PLAY Channel, Noise, Envelope, Period

Channel and Noise will affect the behaviour of any **SOUND** commands; Envelope and Period affect only those **SOUND** commands which have their Volume parameter set to 0. The use of each of the four parameters is described below.

Channel (possible values 0 to 7) selects which of the three tone channels are to be used. Any combination of the three channels is allowed. To see the effect of the channel parameter, try changing line 40 of the last program to

```
40  PLAY 6,0,0,0
```

This will select only channels 2 and 3. Channel 1 is not heard.

The way in which the channels are controlled by the value of the channel parameter is shown in this table.

PARAMETER	CHANNEL
0	All OFF
1	1 ON
2	2 ON
3	1 and 2 ON
4	3 ON
5	1 and 3 ON
6	2 and 3 ON
7	1, 2 and 3 ON

Noise (0 to 7) allows 'white noise' to be added to any or all of the tone channels. In the example we used no white noise. Try changing line 40 to:

```
40 PLAY 7,1,0,0
```

This plays all 3 tone channels and mixes white noise with channel 1. The way in which mixing of white noise is controlled is shown in this table.

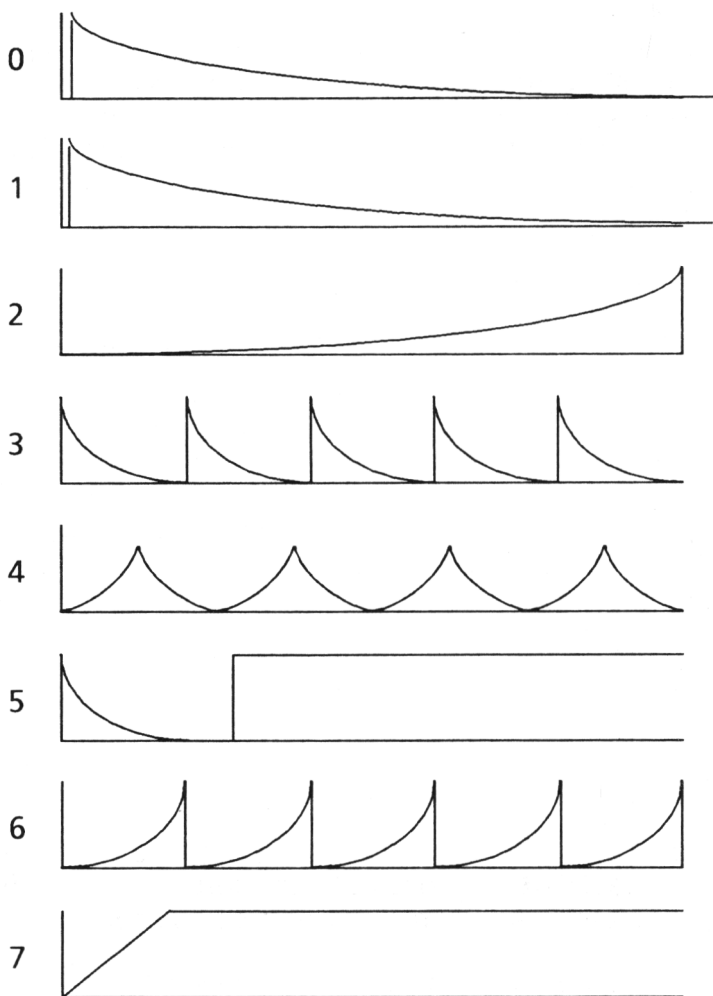
NOISE VALUE	CHANNELS WITH NOISE
0	None
1	1
2	2
3	1 and 2
4	3
5	1 and 3
6	2 and 3
7	1, 2 and 3

Envelope (0 to 7) and Period (0 to 65535) control the way in which the volume of the sound alters in time, or the 'shape' of the sound. The ORIC will produce seven different envelope shapes, which are illustrated on the next page.

Each envelope controls the way in which the volume of the tone varies with time. The best way to find the effect each envelope has on a tone is with the aid of a short program.

```
10 INPUT "WHICH ENVELOPE (0-7)"; E
20 IF E<0 OR E>7 THEN 10
30 INPUT "PERIOD (0-65535)"; P
40 IF P<0 OR P>65535 THEN 30
50 CLS
60 PRINT "ENVELOPE "; E,"PERIOD ";P
70 SOUND 1,1000,0
80 PLAY 1,0,E,P
```

Lines 10 to 40 of this program input an envelope and a period, checking that they are within the allowed range. Lines 50 and 60 clear the screen and display the values you have entered.



ENVELOPE SHAPES

Line 70 creates a tone of period 1000 on channel 1, and because the volume parameter is 0, control of the tone volume is passed to the **PLAY** command in line 80. This selects channel 1, without white noise, and uses the envelope (E) and period (P) you entered.

The **PLAY** command can also be used to increase the flexibility of the **EXPLODE**, **PING** and **SHOOT** commands. If you follow one of these commands with a suitable **PLAY** command you can extend and modify the sound they produce. For example, the program:

```
10 EXPLODE : PLAY 7,7,1,60000
```

will produce a long drawn out explosion, while

```
20 PING : PLAY 1,0,6,50
```

gives a repeating **PING**.

As with all these commands, the **PLAY** command has so many possibilities, that experimenting is necessary to achieve the effect you want.

The third command associated with sound generation is **MUSIC**.

MUSIC

The **MUSIC** command is followed by four parameters, separated by commas, to specify:

- 1 Which tone channel is to be used.
- 2 In which octave the note should be played: there are eight possible octaves numbered 0 to 7, with 0 as the lowest.

- 3 The note which is to be played. There are twelve notes in each octave, numbered 1 to 12.
- 4 The volume at which the note is to be played.

A typical example of the **MUSIC** command would be:

```
MUSIC 1,0,7,5
```

which plays note 7 (F#) from the lowest octave, on channel 1 at volume 5.

The **MUSIC** parameters work as follows:

MUSIC Channel, Octave, Note, Volume

Channel (possible values 1 to 3) selects which of the three tone channels is to be used. Note that this differs from the **SOUND** command channel parameter in that there is no facility for mixing noise.

Octave (0 to 7) selects from which of eight octaves the note is to be played. 0 is the lowest octave, 7 the highest.

Note (1 to 12). The notes in an octave and the numbers that represent them are as follows:

NOTE	NUMBER
C	1
C#	2
D	3
D#	4
E	5
F	6
F#	7

G	8
G#	9
A	10
A#	11
B	12

Volume (0 to 15) controls the volume of the note, 15 being the loudest. If the volume parameter is set to 0, the volume is controlled by the envelope and period parameters of a **PLAY** command.

The **MUSIC** command makes it very easy to enter tunes from sheet music into the ORIC, to make it play them for you.

This program will play a chromatic scale of eight octaves.

```
10  FOR O = 0 TO 7
20  FOR N = 1 TO 12
30  MUSIC 1,O,N,7
40  WAIT 50
50  NEXT N,O
```

As with the **SOUND** command, chords and harmonies can be created by playing different notes on each channel. In order to do this, the **MUSIC** commands must be followed by a **PLAY** command to select the channels. For example, this program plays a different note on each channel and then combines them in a chord:

```
10  MUSIC 1,4,1,7
20  MUSIC 2,4,4,7
30  MUSIC 3,4,8,7
40  FOR C = 1 TO 4
50  IF C = 3 THEN C = 4
60  PLAY C,0,0,0
70  WAIT 100
80  NEXT C
90  PLAY 0,0,0,0
```



```
100  WAIT 100
110  PLAY 7,0,0,0
```

Lines 10 to 30 set up a different note on each channel, each channel is played in turn with a pause in line 70. Line 50 prevents channel 1 and 2 being played together. Line 90 turns off all channels and after a short delay all three channels are played together.

Writing each note in a different **MUSIC** command would be a very cumbersome way of getting the ORIC to play tunes, and to do this more efficiently we will consider different methods of storing data for notes.

PLAYING TUNES

A convenient way of playing simple tunes is to store all the necessary note parameters in a string, reading off one character at a time.

This technique is illustrated in the next program, which plays a short series of notes.

```
10  A$ = "12345678987654321"
20  FOR I = 1 TO LEN(A$)
30  N = ASC(MID$(A$,I,1)) - 48
40  MUSIC 1,4,N,7
50  WAIT 20
60  NEXT I
70  PLAY 0,0,0,0
```

Line 10 contains the data for the notes (remember 0 is not allowed). The data is extracted and played one character at a time by the **FOR...NEXT** loop, within which line 30 successively extracts a single character from A\$, converts it to its ASCII code and subtracts 48 to obtain the correct number. This is then used as the note parameter in the **MUSIC** command in line 40.

This method has the disadvantage that only notes 1 to 9 can be used. You can overcome this by using letters instead of numbers in the string. For example, we could use the first twelve letters of the alphabet to represent the twelve notes, as in this example:

```
10  A$ = "ACEFHJLJHFECA"  
20  FOR I = 1 TO LEN(A$)  
30  N = ASC(MID$(A$,I,1)) - 64  
40  MUSIC 1,4,N,7  
50  WAIT 10  
60  NEXT I
```

There is still a limitation to this method: all the notes are the same length. This is not much use for anything but the simplest music. There are several ways of overcoming this: for example a second string could be used to hold the data for the **WAIT** between notes. Add these lines to the last program to see this technique in operation:

```
15  W$ = "2424242424242"  
35  W = ASC(MID$(W$,I,1)) - 48  
50  WAIT 10 * W
```

This method is acceptable for simple tunes using only one channel. For three part harmony it's better to store all the information for notes and note length in **DATA** statements.

LAND OF HOPE AND GLORY

This section demonstrates one way of converting a musical score into a program to enable the ORIC to play the music. It is not the only way of doing it; you can use your imagination to think of others.

```
5    CLS  
10.  READ A,B,C,T
```

```
20  IF A=0 THEN PLAY 0,0,0,0 :  
    END  
30  PLAY 0,0,0,0  
40  PLAY 7,0,0,0  
50  MUSIC 1, INT(A/12), A -  
    INT(A/12) * 12+1,10  
60  MUSIC 2,INT(B/12),B-  
    INT(B/12)*12+1,8  
70  MUSIC 3,INT(C/12),C-  
    INT(C/12)*12+1,8  
80  WAIT 10*T  
90  GOTO 10  
  
1000 DATA 43,38,35,4, 43,38,35,4  
1010 DATA 42,38,35,1, 43,38,35,1,  
    45,38,35,4  
1020 DATA 40,36,33,10, 38,35,31,10  
1030 DATA 36,31,28,4, 36,31,28,4  
1040 DATA 35,31,28,1, 36,31,28,1,  
    38,31,28,4  
1050 DATA 33,30,24,10, 33,30,26,10  
1060 DATA 35,31,28,10, 37,31,28,1,  
    38,31,28,1, 40,37,33,4  
1070 DATA 45,42,38,10, 38,33,30,10  
1080 DATA 43,38,35,4, 43,38,35,4  
1090 DATA 43,38,35,1, 42,38,35,1,  
    40,38,35,4, 38,33,30,22  
1100 DATA 43,38,35,4, 43,38,35,4  
1110 DATA 42,38,35,1, 43,38,35,1,  
    45,38,35,4  
1120 DATA 40,36,33,10, 38,35,31,10  
1130 DATA 36,31,28,4, 36,31,28,4  
1140 DATA 35,31,28,1, 36,31,28,1,  
    38,31,28,4  
1150 DATA 33,30,24,10, 33,30,26,10  
1160 DATA 35,31,28,10, 37,31,28,1,  
    38,31,28,1, 40,37,33,4  
1170 DATA 45,42,38,10, 38,33,30,10  
1180 DATA 48,42,38,10, 48,42,38,1,  
    47,42,38,1, 45,42,38,4
```

```
1190 DATA 47,47,38,4, 47,43,38,4,  
        47,42,38,4, 47,41,38,4  
1200 DATA 40,36,33,10, 42,36,33,1,  
        43,36,33,1, 45,36,33,4  
1210 DATA 38,35,31,10, 43,38,31,10  
1220 DATA 31,28,24,4, 31,28,24,5  
1230 DATA 36,31,28,1, 35,31,26,6  
1240 DATA 33,30,26,3, 31,26,19,22  
6000 DATA 0,0,0,0
```

The data for the tune is contained in lines 1000 to 6000. As you can see, the data items are arranged in groups of four. Take as an example line 1000:

```
1000 DATA 43,38,35,4 43,38,35,4
```

The first three items in each group contain information about the notes on each of the three channels, and the octaves in which they are played. The final item in the group determines the length of time for which the notes are to be played.

Line 10 reads the data into the variables A, B, C and T. A holds the data for channel 1, B for channel 2 and C for 3, and T is the time.

Line 30 turns off the current sound.

Line 40 turns on all three channels, with no noise or envelope.

Lines 50 to 70 convert the data in A, B and C into octave and note values. Channel 1 is slightly louder than the others because it is sounding the melody.

If you do not understand the way music is written on paper, there is a brief guide in Appendix 9 of this book.

SUMMARY

In addition to the programmed sound commands **ZAP**, **EXPLODE**, **SHOOT** and **PING** there are three BASIC commands which are used to control sound generation.

SOUND C,P,V generates a tone of period P, at volume V, using a combination of the three channels controlled by the parameter C.

PLAY C,N,E,P controls the tone channels and their mixing with the noise channels using the C and N parameters, and provides a choice of envelope shape and period with the E and P parameters. Used in conjunction with **MUSIC** and **SOUND** commands.

MUSIC C,O,N,V allows generation of pure musical notes. The note N from octave O is played at volume V on the channel determined by C.

NOTE You may have noticed that some of the information given in this chapter differs from that given in the manufacturer's manual. Where this occurs, it is *this* book that is correct.

CHAPTER 11

CHARACTERS

This chapter deals with the way in which the ORIC identifies, stores and makes use of the various letters, numbers and symbols it is able to display.

To demonstrate this the following short program detects your key presses and displays the corresponding character on the screen together with the ASCII code for that character. You can stop the program by pressing the space bar.

```
10   CLS
20   GET K$
30   PRINT K$, ASC(K$)
40   IF K$=" " THEN END ELSE 20
```

The ORIC can handle 256 different characters, each of which has a code number between 0 and 255 - the ASCII code. A list of characters and their codes is given in Appendix 4.

You will notice that not all of the 256 possible codes are listed in the appendix. This is because not all of them are used to represent letters and numbers. The table below shows how the codes are allocated.

You can see that there are different types of character. Let's first examine how the ORIC handles letters and numbers.

ASCII CODE	USE
0 - 31	Control Codes/Attributes
32 - 127	Standard Character Set
128 - 159	Attributes
160 - 255	Alternate Character Set

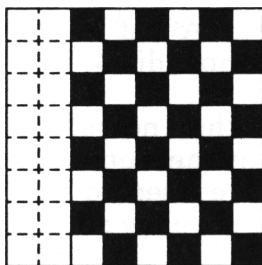
ASCII Code allocation

ALPHANUMERIC CHARACTERS

In text mode, type

```
PRINT CHR$(126)
```

This will display character 126, which is a chequered square, on the screen. If you look closely at the character, you will see that it is composed of tiny dots, called **pixels** (picture elements). We can represent the character on an 8 x 8 grid like this:



Notice that the two lefthand columns of the grid aren't used to define the character.

This grid pattern illustrates the way in which the ORIC stores data representing the alphanumeric

characters. Each row of the grid represents one memory location, and each block on a row represents one bit in that memory location. For an illuminated pixel, the corresponding bit in memory is set to 1, otherwise to 0. The diagram shows character 126 again, this time with the values of each bit of the memory location written along the top.

128	64	32	16	8	4	2	1	
								42
								21
								42
								21
								42
								21
								42
								21

We can calculate the values which must be stored in the block of 8 memory locations to define the character by adding up the numbers for each illuminated pixel. These numbers are shown on the right of the grid. (If you are not familiar with binary numbers read Appendix 8 to make this clear).

To check this, let's look at the memory locations in which the ORIC stores the data for character 126.

When you switch on your ORIC, data representing the characters it displays is transferred from ROM

into RAM. The area of RAM used depends upon whether TEXT or HIRES mode has been set (see Appendix 7 to find the address of this area). In TEXT mode the character data is stored between 46080 and 48000 (13312 to 15232 for a 16K machine). If the character with code 1 is stored at 46080, and each character takes a block of 8 memory locations of storage, you can then calculate where in memory any character is stored. To locate the block of eight memory locations defining the character 126, take its ASCII code, multiply it by 8, then add this number to the start address of the character set memory: i.e.

$$(8 * 126) + 46080 = 47088$$

To examine the contents of this area of memory, use this program

```
10 FOR A = 47088 TO 47095
20 PRINT PEEK (A)
30 NEXT A
```

(PRINT PEEK(A) prints the value found in memory location A. PEEK is explained in chapter 15.)

The same pattern of numbers as we calculated for character 126 is displayed.

You can do this for any character:

```
10 ST = 46080
20 INPUT "ASCII CODE (32 - 127)";C
30 CA = ST+(8 * C)
40 PRINT "MEM LOC  CONTENTS":PRINT
50 FOR A = CA TO CA+7
60 PRINT A, PEEK (A)
70 NEXT
```

DESIGNING YOUR OWN CHARACTERS

Because the character data is stored in RAM, you can alter it to suit your needs. For example, enter ASCII code 49 - the code for the character 1 - in the above program. Now, let's change the 1 character by turning it upside down. The data for the 1 character, obtained from running the last program, is

46472								8
46473	+							24
46474	+							8
46475	+							8
46476	+							8
46477	+							8
46478	+							28
46479	+							0

Notice that the data for the bottom line is 0, to allow for a space between lines on the screen. To turn the 1 upside down, type in the following

```
POKE 46472, 28
POKE 46473, 8
POKE 46474, 8
POKE 46475, 8
POKE 46476, 8
POKE 46477, 24
POKE 46478, 8
```

The POKE command is explained fully in chapter 15. What you have done is replace the contents of the POKEd locations with the new values (which are the old values in reverse order).

From now on, until you switch off the machine, reset or redefine that character, all 1s will be displayed upside down.

This is a very laborious way of redefining characters, a job which you can use the ORIC to help you with. The following program allows you to redefine any character by moving the cursor around a large picture of that character, changing pixels as required.

CHARACTER GENERATOR

```
100 DIM A%(8,9)
110 CLS
120 INPUT "CHARACTER CODE (32-
    125)"; C
130 IF C < 32 OR C > 125 THEN
    PING: GOTO 110
140 ST = 46080 + (C*8) 'START OF
    CHARACTER DEFINITION IN RAM
150 SC = 48335 : CC = 48378
160 CLS: PRINT CHR$(17)
170 FOR R = 1 TO 8: GOSUB 1000:
    NEXT R
180 GOSUB 2000
185 IF F=1 THEN END
190 X=3: R=1
200 B=PEEK(CC)
220 GOSUB 3000
225 CC = SC + X + 40 * R
230 GOTO 200

1000 REM STORE CHARACTER DATA IN
    ARRAY
1010 N=PEEK(ST+R-1): X=8
1020 A%(R,9) = N
1030 REPEAT
1040 : IF N/2 = INT(N/2) THEN
    A%(R,X)=0 ELSE A%(R,X)=1
```

```
1050 : N = INT(N/2): X=X-1
1060 UNTIL N<1
1070 FOR I=X TO 1 STEP -1
1080 A$(R,I) = 0
1090 NEXT I
1100 RETURN

2000 CLS
2010 FOR R=1 TO 8
2020 FOR X=1 TO 8
2030 IF A$(R,X)=1 THEN CS=128 ELSE
      CS=126
2040 POKE SC+(X+(R*40)),CS
2050 NEXT X
2060 PLOT 24, R+7, STR$(A$(R,9))
2070 NEXT R
2080 RETURN

3000 REM MOVE CURSOR AND CHANGE
      PIXELS
3010 REPEAT
3020 : K$ = KEY$
3030 : CH=PEEK(CC): C%=(CH OR
      128)-(CH AND 128) 'FLASH
      CURSOR
3040 : POKE CC, C%: WAIT 20
3050 UNTIL K$ <>" "
3060 IF K$=CHR$(27) THEN PRINT
      CHR$(17): POP: GOTO110 'ESC
      KEY
3070 IF K$=CHR$(8) AND X > 3 THEN
      POKE CC,B: X=X-1: RETURN
      'LEFT
3080 IF K$=CHR$(9) AND X < 8 THEN
      POKE CC,B: X=X+1: RETURN
      'RIGHT
3090 IF K$=CHR$(10) AND R < 8 THEN
      POKE CC,B: R=R+1: RETURN
      'DOWN
3100 IF K$=CHR$(11) AND R > 1 THEN
      POKE CC,B: R=R-1: RETURN 'UP
```

```
3110 IF K$=" " THEN A%(R,X) = 1-  
      A%(R,X): GOTO 3200 'CHANGE  
      PIXEL  
3120 IF K$="P" THEN 4000  
3130 PING: RETURN  
3200 IF B=128 THEN POKE CC,126:  
      RETURN  
3210 POKE CC,128: RETURN  
  
4000 M$=CHR$(12)+"REPROGRAMMING  
      CHARACTER"+STR$(C)  
4010 PLOT 5,22,M$  
4020 FOR R = 1 TO 8  
4030 FOR X = 1 TO 8  
4040 D = D + (2^(8-X)) * A%(R,X)  
4050 NEXT X  
4060 POKE ST+R-1, D  
4070 D=0: NEXT R  
4080 CLS: PRINT CHR$(17)  
4090 PLOT 14, 11, "COMPLETED"  
4100 END
```

Apart from creating 'Space Invader' characters and the like, one application of redefined characters is to create a reverse video character set. The following short program will reprogram the lower case letters to be reverse video upper case.

```
10 FOR A= 46856 TO 47063  
20 POKE A,255-PEEK(A-32*8)  
30 NEXTA
```

Should you wish to save a character or set of characters you have created, you may do so by following the instructions in Chapter 14.

INVERSE CHARACTERS

You will recall that the ORIC uses only six bits of every eight to define its characters - reserving the other two for its own use.

If bit 7 is set to 1, the ORIC will display the character in **inverse**, i.e. in the opposite colour to the background. Try this program to demonstrate this.

```
10    CLS
20    A$ ="INVERSE CHARACTER DEMO"
30    FOR I=1 TO LEN(A$)
40    B$ = B$ + CHR$ (ASC (MID$
      (A$,I,1)) + 128)
50    NEXT
60    PLOT10,10,A$
70    PLOT10,12,B$
80    FOR I=1 TO 7
90    PAPER I
100   WAIT 100
110   NEXT
```

Line 40 creates a string B\$ to be the same as A\$, but adds 128 to the ASCII code of each character, to set bit 7 to 1. Notice that the background colour of the lower of the two strings (B\$) changes to the inverse of the screen background each time the **PAPER** colour changes (in the same way as the cursor).

NOTE You cannot use **PRINT** statements to display such inverse strings because the **PRINT** command clears Bit 7 to 0. You can, however, use **POKE** to put inverse characters into screen memory.

CONTROL CHARACTERS

The ORIC reserves the ASCII codes 0 to 31 to represent special characters known as control characters. These characters cannot be seen on the screen, but their effect can. For example, to clear the screen, you have used the key sequence CTRL and L. This instructs the ORIC to 'print control

character 12'. You could achieve the same effect by typing

```
PRINT CHR$(12)      (try it!)
```

A complete list of the control characters and their effect is given in Appendix 5.

ATTRIBUTES

The ORIC also interprets as special the characters which control the colour and appearance of the display, known as Attributes. As with control characters, you can't see them, but you can see their effect. A full list of attributes is given in Appendix 3.

COLOUR ATTRIBUTES

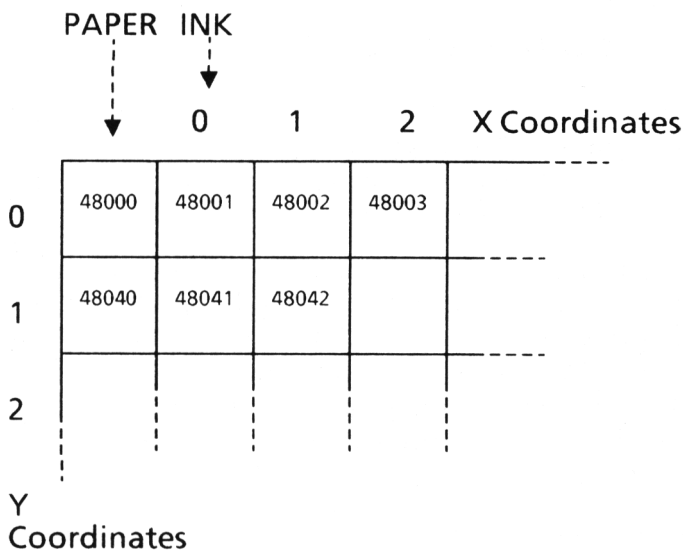
The ORIC uses some of the attributes to control the colour of its display.

The two lefthand columns of the screen are reserved to contain attributes controlling the foreground and background colours of the screen. The extreme lefthand column contains characters which determine the background or **PAPER** colour. Column 0 controls the foreground or **INK** colour.

The diagram on the next page shows the layout of the top lefthand corner of the screen in **TEXT** mode (the number in each box is the memory address of that location).

When the ORIC executes a **PAPER** statement, it places the code for that **PAPER** colour in every square in the left hand column.

We can see this happen using a program.



Screen Layout

```

10 CLS
20 INPUT "PAPER COLOUR (0-7)";P
30 PAPER P
40 FOR I=48040 TO 49040 STEP 40
50 PRINT PEEK(I)
60 NEXT I

```

The program uses the **PEEK** command to examine every memory location in the lefthand column. If you **RUN** the program, you will see the screen change to the selected colour, and the contents of the lefthand column will be displayed.

For example, **PAPER 4** turns the screen blue, and the number 20 is found in the **PAPER** column. This is because the ORIC interprets character 20 as the attribute for blue paper.

We can use the same technique to investigate the **INK** attributes in column 0

```
10 CLS
20 INPUT"INK COLOUR (0-7)";I
30 INK I
40 FOR J=48041 TO 49041 STEP 40
50 PRINT PEEK(J)
60 NEXTJ
```

A colour attribute determines the **INK** or **PAPER** colour for the rest of the row to its right, or until a further attribute is found. We can set foreground and background attributes for individual rows on the screen. To do this we will need to know the attribute codes for each colour. These are given in the following table.

COLOUR	FOREGROUND	BACKGROUND
BLACK	0	16
RED	1	17
GREEN	2	18
YELLOW	3	19
BLUE	4	20
MAGENTA	5	21
CYAN	6	22
WHITE	7	23

Colour Attribute Codes

This program sets foreground and background colour for each row to a random value. **RUN** the program and then **LIST** it to see the effect.

```
10 CLS
20 X=48000
30 REPEAT
40 : I=INT(RND(1)*8)
50 : P=INT(RND(1)*8)+16
60 : X=X+40
70 : POKE X,P:POKE X+1,I
80 UNTIL X=49080
```

You can place colour attributes anywhere on the screen, where they occupy a character space and control the foreground or background display of all the characters to their right.

```
10 CLS
20 REPEAT
30 S=INT(RND(1)*1000)+48040
40 P=INT(RND(1)*8)+16
50 POKE S,P
60 UNTIL KEY$=" "
```

In this case, S is a random screen position, and P is a random **PAPER** colour. The program will continue until you press the space bar.

It is not necessary to **POKE** colour attributes on to the screen. **PLOT** will work just as well.

```
10 CLS
20 REPEAT
30 X=INT(RND(1)*38)
40 Y=INT(RND(1)*27)
50 A=INT(RND(1)*8)+16
60 PLOT X,Y,A
70 UNTIL KEY$=" "
```

FLASHING & DOUBLE HEIGHT CHARACTERS

The remainder of the attributes are used to control the appearance of the display. You can make characters on the screen flash, or display them in double height by placing the appropriate attribute character in the location before the character you want to affect. The attribute will affect all those characters to its right, up to the edge of the display, or until another attribute cancels its effect.

The codes for these attributes are given in the table. (The alternate character set is explained shortly).

CODE	EFFECT
8	Normal height, Standard character set
9	Normal height, Alternate character set
10	Double height, Standard character set
11	Double height, Alternate character set
12	Normal height, Standard character set, flashing
13	Normal height, Alternate character set, flashing
14	Double height, Standard character set, flashing
15	Double height, Alternate character set, flashing

Character Attribute Codes

Here is an example of the use of these attributes.

```
10 PAPER 1
20 A$=CHR$(7) + "WHITE LETTERS"
30 PLOT 10,10,A$
```

This will display the message in the middle of the screen. Notice that the attribute has a different effect if you instruct the ORIC to

```
PRINT A$
```

This is because the use of the **PRINT** command tells the ORIC to interpret the CHR\$(7) as a **control character**, not an attribute. (Though in some cases it treats them as both !)

If you want to use attributes with the **PRINT** command, you must add 128 to the code you require (i.e. set bit 7 to 1). So

```
10 A$=CHR$(12+128)+"FLASHING TEXT"
20 PRINT A$
```

is another way of using attributes. However, if you now try to **PLOT A\$**

```
30 PLOT 10,10,A$
```

you will see a square appear in front of the flashing message. This square is an inverse space character and will be in the inverse colour to the background. This is because, in setting bit 7 to prevent the attribute code being interpreted as a control character, you have also instructed the ORIC to print that character in inverse (as discussed earlier)! This is a bit confusing, but it's really just a matter of choosing the right attribute codes for the display command you want to use.

128 - 160	with POKE and PLOT will give inverse effect
128 - 160	for PRINT
0 - 32	with PRINT will act as control characters

DOUBLE HEIGHT CHARACTERS

You can tell the ORIC to print double height characters in much the same way as flashing characters, except that you must first have that message displayed on two consecutive lines on the screen. You can get the ORIC to do this for you by using a control character. Try pressing CTRL and D (to print control character 4); everything you now type will appear on two lines at once. To turn off this effect, type CTRL and D again. We can obtain double height characters directly from the keyboard as follows.

First move the cursor away from the edge of the screen by pressing the space bar, and press CTRL and D together, followed by ESC and J (this tells the ORIC to print attribute 10, the one for double height graphics). Now type some characters and they will appear twice their normal size.

NOTE You must start your message on an even numbered line (0,2 etc), or your double height characters will appear with the top half below the bottom half!

To obtain double height characters in a program, use a line such as

```
10 PRINT " CHR$(4)CHR$(27)"J
   YOUR MESSAGE"CHR$(4)
```

In this statement, the space moves the cursor away from the left hand edge of the screen (where CHR\$(4) would act as the PAPER attribute for that

line), CHR\$(4) switches on double printing, CHR\$(27) is the ESC character and J is the attribute for double height text. Remember to turn off the double printing by adding another CHR\$(4) at the end of your double height message.

ALTERNATE CHARACTER SET

The codes 160 to 255 are reserved for the Alternate Character Set. This is a set of characters designed to give displays similar to those of PRESTEL and CEEFAX services; it is described fully in the next chapter.

SUMMARY

Characters are represented in the ORIC's memory by numbers. The numbers representing each character correspond to the ASCII code.

The character shapes displayed on the screen are defined by blocks of eight memory locations. The contents of these memory locations may be altered to define new characters.

Certain of the possible character numbers do not have a corresponding display symbol. These numbers are used for **control characters** which control the behaviour of the display.

The colour and other properties of characters displayed on the screen are controlled by further characters called attributes, which when printed alter the properties of the characters to the right of them on the same line.

CHAPTER 12

LOW RESOLUTION GRAPHICS

In addition to displaying text on the screen, the ORIC can generate and display a variety of graphics shapes and symbols.

There are two distinct types of graphics - Low Resolution and High Resolution. The difference between the two is in the size of the individual blocks, or **pixels**, (picture elements) which make up the display, and in the amount of memory used by the display.

LOW RESOLUTION GRAPHICS

Low resolution graphics use the same screen layout as text mode - 27 rows of 40 columns. In low resolution mode, or Lores, each square on the screen which contains a letter or number in TEXT mode can contain either a letter or number, or a Lores graphics character.

To see the range of Lores characters, type in and **RUN** this program.

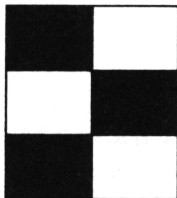
```
5  CLS
10 LORES1
20 FOR I = 32 TO 95
30 PRINT CHR$(I);SPC(6);
40 NEXT I
```

The Lores graphics on the ORIC are of the same type as those used for viewdata services such as PRESTEL, CEEFAX and ORACLE. They are built up from a grid composed of six blocks.

1	2
4	8
16	32

The Lores character grid

Each block in the character is assigned a number, and these are used to determine the character code. For example, suppose we wanted to display a character which looked like this:



To find the code for this character, first add up the numbers of the shaded squares

$$1 + 8 + 16 = 25$$

Then add 32

$$25 + 32 = 57$$

This is the ASCII code for the character '9' (see Appendix 4).

In order to display our graphics character, we need to put character 9 on the screen. However, we must also specify that we require a graphics character, and not the number '9'. There are two ways to do this.

In TEXT mode:

```
Clear the screen (CTRL L)
Move the cursor in from the left with the
space bar
Press ESC, then I
Now press 9
```

Our graphics character is displayed.

The ESC I sequence instructed the ORIC that the next character should be the graphics character corresponding to '9', rather than the alphanumeric character.

We can use the same technique within a program :

```
10 CLS
20 PRINT "CHR$(27)"I9"
```

Here we use CHR\$(27) to represent the 'ESC' character because we cannot type the 'ESC' character into a program.

You can use this short program to try out other characters

```
10 CLS
20 INPUT "ASCII CODE";C
30 PRINT C;CHR$(27);"I";CHR$(C)
40 GOTO 20
```

As you can see this method allows both alphanumeric and Lores graphics characters on the same line.

Depending upon when you bought your ORIC, you may find that the LORES characters are a bit lopsided, because the pixels are not all the same size. This is a fault of the ORIC, and to get around it you will either have to design your low resolution displays accordingly, or reprogram the alternate character set. This short program will do the job.

```
10 FOR I = 47368 TO 47858
20 IF PEEK(I)=240 THEN POKE I,56
30 IF PEEK(I)=15 THEN POKE I,7
40 NEXT
```

You could include this program as a subroutine in programs that use Lores graphics characters.

There is another way of using Lores graphics, which was used in the first example program - the BASIC command **LORES**.

LORES

LORES is followed by a parameter N, which can be either a 0 or a 1. A 0 means that all characters printed on the screen will be from the standard character set; a 1 selects the alternate set, which contains the 64 Lores characters.

The following program illustrates the difference between the two by first writing a message using the standard set, then switching to the alternate set and rewriting the message.

```
5 CLS : A$ = "CHARACTER SET DEMO"
10 LORES0
20 PRINT
30 PRINT A$
```

```
40 WAIT 500
50 LORES1
60 PRINT
70 PRINT A$
80 WAIT 500
90 CLS
```

Each time a **LORES** command is executed, the screen is cleared to black, and every subsequent **PRINT** or **PLOT** statement is executed using the specified character set, until another **LORES**, **TEXT** or **CLS** statement is executed.

MIXING CHARACTER SETS

It is possible to use characters from both character sets on the screen at the same time, by sandwiching the 'foreign' character or string with **CHR\$(8)** and **CHR\$(9)**, as in this example:

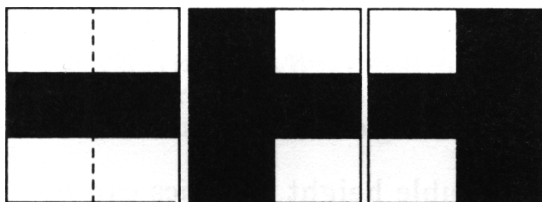
```
10    LORES0
20    M$ = "LORES 0 SELECTED"
30    U$ =
      CHR$(9)+"#####"
      +CHR$(8)
40    PLOT 12,5,M$
50    PLOT 11,6,U$
60    PLOT 1,9,"UNDERLINING USING
      ALTERNATE CHARACTERS"
```

This technique works in reverse: you can display alphanumerics in **LORES 1**:

```
10    LORES1
20    U$ = CHR$(8)+"TEXT IN LORES1"
      +CHR$(9)
30    M$ = "NANANANA"
40    FOR Y=0 TO 24
50    PLOT 16,Y,M$
60    NEXT Y
70    PLOT 12,12,U$
```

As we saw in the previous example, Lores graphics characters can be assembled into strings in the same way as text characters, and we can use this method to draw lines and patterns.

For example, we can draw a grid pattern on the screen using the three characters shown below:



First, find the character codes by adding up the numbers of the shaded squares and adding 32:

$4 + 8 = 12$	$12 + 32 = 44$	'.'
$1 + 4 + 8 + 16 = 29$	$29 + 32 = 61$	'='
$2 + 4 + 8 + 32 = 46$	$46 + 32 = 78$	'N'

We build up a string containing the characters
'.=N'

$$A\$ = ",=N"$$

For use in TEXT mode, the string must begin with a space, followed by the 'ESC' character and an 'ESC' code

Our string becomes

$$A\$ = " "+CHR\$(27)+"I"+",=N"$$

We can now PRINT the expanded string.

$$10 \quad A\$ = " "+CHR\$(27)+"I"+",=N,=N,=N,=N,=N,=N,=N,=N,=N"$$

20 PRINT A\$

ESCAPE CODES

In the same way that we obtained double height and flashing characters in Chapter 11, so can we with Lores graphics.

If we alter line 10 of our example to read

```
10  A$ = " " + CHR$(27) + "M" +
      ",=N,=N,=N,=N,=N,=N,=N,=N,=N"
```

our graphics string can be made to flash.

Similarly, double height graphics can be obtained by replacing the M in line 10 with a K, and adding CHR\$(4) (which is CTRL D) to the string

```
10  A$ = " "+CHR$(4)+CHR$(27)+"K"
      + ",=N,=N,=N,=N,=N,=N,=N,=N,=N"+CHR$(4)
```

Remember that double height characters require accurate vertical positioning - to prevent them appearing with the top half of the character below the bottom half, you must position the string on an even numbered line.

To make the double height characters flash, replace the 'K' with an 'O'.

MOVING GRAPHICS

We can use the PLOT command to place graphics characters anywhere on the screen in LORES1 mode. This program creates a string of characters, and moves them slowly across the screen.

```
10  LORES1
20  A$="XU"
30  FORX=1 TO 36
```

```
40    PLOTX,10,A$
50    PLOTX-1,10," "
60    WAIT10
70    NEXTX
```

COLOUR WITH LORES GRAPHICS

In previous chapters, we learned how to change foreground and background colours with the **INK** and **PAPER** commands.

This works equally well when **PRINTing** graphics strings in **TEXT** mode. Since **INK** and **PAPER** are *global* commands, the entire screen will be affected.

However, these commands are not as useful in **LORES** modes. You may change the colour of the characters from either character set, using **INK**, in either **LORES** mode, but **PAPER** doesn't work: the screen is always set to black.

This short program plots a graphic string in **LORES 1** and displays all the **INK** colours.

```
10    LORES1
20    PLOT15,15,"ABCDEFGG"
30    FOR I = 0 TO 7
40    INK I
50    WAIT 100
60    NEXT I
70    WAIT 100
80    CLS
```

If we change line 40 to read

```
40    PAPER I
```

you will notice that only one of the screen columns adopts the **PAPER** colour - the rest of the screen remains black. This is because of the way in which

th ORIC controls its screen display - with the use of attributes. You can make use of attributes in LORES modes in exactly the same way as described for TEXT mode in Chapter 11.

This program uses most of the ideas in this chapter to plot a histogram (bar chart), showing the spread of numbers generated by the RND command.

```
5      INK7:PAPER0
10     LORES1
20     DIMA%(40)
30     PRINTCHR$(17)
40     PRINTCHR$(20)
50     Y$ = "5"
60     X$ = "#"
70     P$ = "f"
80     T1$ = CHR$(8) + "Histogram
        showing Random Numbers" +
        CHR$(9)
90     T2$=CHR$(8)+"between 0 and 35
        being generated"+CHR$(9)
100    N$=CHR$(8)+" 0
        Number
        35"+CHR$(9)
110    GOSUB 500
120    GOSUB 200
130    FOR C=1 TO 300
140      : N=INT(RND(1)*36)
150      : A%(N)=A%(N)+1
160      : IFA%(N) > 20 THEN A%(N)=20
170      : PLOT N+3,23-A%(N),P$
180    NEXTC
190    GETZ$: PRINT CHR$(17)
        CHR$(20): CLS
195    END
200    FOR Y=3TO23
210      PLOT 1,Y,Y$
220    NEXTY
230    FORX=1 to 38
240      PLOTX,23,X$
```

```
250  NEXTX
260  PLOT2,0,T1$
270  PLOT2,1,T2$
280  PLOT1,24,N$
290  RETURN
500  FORI= 48121 TO 48961 STEP 40
510  POKE I,5
520  POKE I+2,3
530  NEXTI
540  RETURN
```

Lines 5 to 40 set the display mode, turn off the cursor and select lower case (simply to tidy up the display).

Lines 50 to 70 set the characters used for the X and Y axes and the bars of the chart.

Subroutine 500 places attributes on the screen to control the colour of the axes.

Subroutine 200 draws the axes and labels the chart.

Lines 130 to 180 plot the results for the function in line 140 for 300 numbers, with line 160 guarding against any part of the histogram overwriting the titles.

SUMMARY

There are two low resolution graphics modes, designated LORES0 and LORES1, which display characters from the standard and alternate character sets respectively on a black screen.

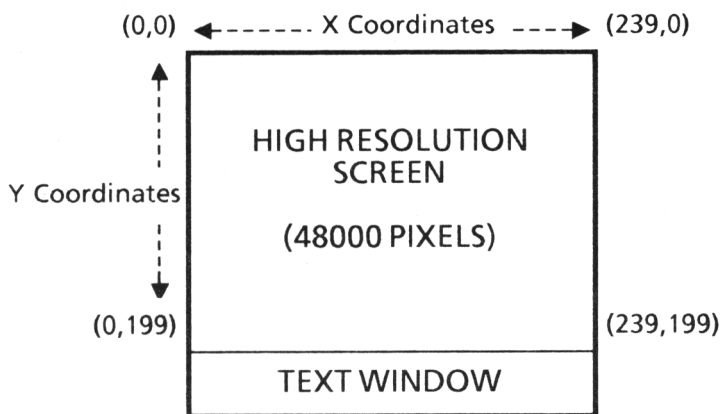
Characters from both character sets can be displayed in either mode by sandwiching the 'foreign' characters with CHR\$(8) and CHR\$(9).

The LORES screen has the same layout as the TEXT screen, and its appearance can be controlled using attributes in the same way as the TEXT screen.

CHAPTER 13

HIGH RESOLUTION GRAPHICS

In high resolution mode, the ORIC screen is divided into two sections, an area 200 rows by 240 columns, upon which high resolution graphics may be displayed, and a small **Text Window**, which displays three lines of text at the bottom of the screen.



The HIRES Screen

To make use of the high resolution mode, there is a BASIC command, **HIRES**.

HIRES

Typing **HIRES** selects high resolution graphics, turns the **HIRES** screen black, and leaves a three line Text area at the foot of the screen in the latest **PAPER** colour. Try it!

To return to text mode, type **TEXT**.

HIRES mode has its own (normally invisible) cursor, which is set to position (0,0) (the top left-hand corner of the screen) when **HIRES** is selected. All commands in **HIRES** use the position of this cursor as a starting point. For example, let's draw a line from the top left-hand corner of the screen (0,0) to the bottom right-hand corner (239,199). To do this, we use the BASIC command **DRAW**.

DRAW

Select **HIRES** - the cursor is now at (0,0).
Now type

```
DRAW 239,199,1
```

the line is drawn.

The draw command works like this :

```
DRAW X,Y,FB
```

The X and Y parameters are **relative** coordinates, that is they tell the ORIC how many positions across (X) and down (Y) the screen the end of the line is to be, **from the present cursor position**.

In our example, the cursor had been set to (0,0) by the **HIRES** command, so the **DRAW** statement instructed the ORIC to draw a line from (0,0) to (239,199).

The third parameter in the **DRAW** command is the FB (Foreground/Background) number. This tells the ORIC in which colour to draw the line. The possible FB numbers and their meanings are shown in this table.

FB NUMBER	EFFECT
0	Draw line in background colour
1	Draw line in foreground colour
2	Draw line in opposite colour to present colour
3	Null - no line but move cursor

We can think of the HIRES cursor as a pen, whose behaviour is controlled by the FB parameter. The FB numbers 0, 1 and 2 control the colour in which the 'pen' draws the line. If the FB number is 3, the 'pen' is lifted off the 'paper', so no line is drawn, but the 'pen' is still moved. Having drawn the line, the 'pen' waits at the end of that line, for your next instruction.

To check the effect of the FB number, let's draw a line to the top right-hand corner of the screen, in the same colour as the background.

`DRAW 0,-199,0`

Remember that the X and Y parameters are relative to the current cursor position.

Notice that a negative number is used to instruct the ORIC to move back up the screen. The line is drawn, but we can't see it - it's the same colour as the background. A line drawn in the background colour will become visible only when it passes over an area of foreground colour.

The cursor is now at the top right-hand corner (239,0)

As we have seen, the **DRAW** command uses relative coordinates. This can sometimes be unhelpful, because you must always know where the cursor is. One way of using **absolute** coordinates in **DRAW** statements, is to make use of two of the system variables. Locations 536 and 537 contain the X and Y coordinates of the **HIRES** cursor. If we include these in a **DRAW** command, we can easily **DRAW** a line to a given point, from anywhere on the screen.

```
DRAW X-PEEK(537),Y-PEEK(538),1
```

where X and Y are the coordinates of the point you want to draw to.

To get the hang of the **DRAW** command, try experimenting with the parameters. Remember that if you try to make the cursor move outside the screen area, an error message will be displayed.

MOVING THE HIRES CURSOR

There are two **BASIC** commands which give you control over the **HIRES** cursor, **CURMOV** and **CURSET**.

CURMOV X,Y,FB

The parameters in **CURMOV** are identical to those for the **DRAW** command. X and Y are **relative** coordinates, and the FB number is either 0,1,2 or 3. To see **CURMOV** in action, type **HIRES** (the cursor will now be at (0,0)). Now type

```
CURMOV 50,50,1
```

This moves the cursor 50 pixels across and 50 pixels down the screen from the last cursor position. The cursor is visible because we specified **FB = 1**. Type in the command again

```
CURMOV 50,50,1
```

The cursor is now at (100,100).

The other cursor controlling command is **CURSET**.

CURSET X,Y,FB

CURSET differs from **CURMOV** in that the **X** and **Y** parameters are **absolute**. Wherever the cursor happens to be, **CURSET** will move it to the position specified by **X** and **Y**.

There is a corresponding BASIC command, **POINT** which enables you to test any point on the screen to determine its **FB** number.

POINT(X,Y)

```
PRINT POINT (100,100)
```

will return a value of 0 if point (100,100) is in the background colour, and -1 if it is in the foreground colour. You can use this command to detect the change of a pixel from background to foreground colour, brought about by advancing aliens perhaps. However, in many ORICs a bug means that this command can interfere with the commands that move the **HIRES** cursor (**DRAW**, **CURMOV** and **CURSET**), making it rather less useful.

The following program demonstrates the three commands **CURMOV**, **CURSET** and **DRAW** by drawing a familiar object!

```
1000 HIRES
1020 CURSET 17,50,0
1030 DRAW 222,0,1
1040 DRAW 0,140,1
1050 DRAW -222,0,1
1060 DRAW 0,-140,1
1070 CURSET 17,80,1:DRAW 222,0,1
1080 CURSET 17,175,1:DRAW 222,0,1
1100 CURSET 29,92,0
1110 X=11:Y=8
1120 FOR K=1 TO 13 : GOSUB 2000 :
      NEXT
1130 CURSET 23,108,0
1135 FOR K=1 TO 14:GOSUB 2000:NEXT
1140 CURSET 27,124,0
1145 FOR K=1 to 12 :GOSUB
      2000:NEXT
1150 X=23:GOSUB 2000
1160 CURSET 40,140,0
1165 X=11
1170 FOR K=1TO12:GOSUB 2000:NEXT
1180 CURSET 70,156,0
1190 GOSUB 2000:GOSUB 2000
1200 X=71:GOSUB 2000
1210 X=11:GOSUB 2000:GOSUB 2000
1999 END
2000 DRAW X,0,1
2010 DRAW 0,Y,1
2020 DRAW -X,0,1
2030 DRAW 0,-Y,1
2040 CURMOV X+4,0,0:RETURN
```

The program works like this.

Lines 1000 to 1080 draw an outline of the ORIC.

Lines 1100 to 1210 draw the individual keys by repeatedly calling the subroutine at line 2000, which draws a square, the dimensions of which are determined by X and Y. At the end of the

subroutine the cursor is moved to the top left-hand corner of the start of the next key.

After you have typed in and run the program, either save it on tape (see Chapter 14) or leave it in the machine, because we shall refer to it later on in this chapter.

So far, all the lines we have drawn have been solid. The ORIC is capable of drawing dotted lines just as easily. This is achieved with the use of the **PATTERN** command.

PATTERN N

The **PATTERN** command has one parameter, **N**, which is a number between 0 and 255. This number is set to 255 when you enter **HIRES** mode, and remains at 255 until you change it with the **PATTERN** command. We say 255 is the **default** value.

Try the following program:

```
10 HIRES
20 INPUT "PATTERN (0-255)";P
30 PATTERN P
40 CURSET 0,100,0
50 DRAW 239,0,1
60 GETA$
70 GOTO 10
80 END
```

Start with a pattern value of 255, then try others to see their effect on the line.

The type of pattern produced depends upon the pattern of 1s and 0s in the binary form of the pattern number as shown below.

Number	Binary	Pattern
255	11111111	_____
15	00001111	___ ___ ___
170	10101010	-----

Patterns with PATTERN

If you insert the line

```
1005 PATTERN 170
```

in our drawing program, you will create a dotted version of the drawing. Remember that you will have to type `RUN 1000` to run the drawing program, unless you delete the pattern example program (Lines 10 to 80).

CHARACTERS IN HIRES

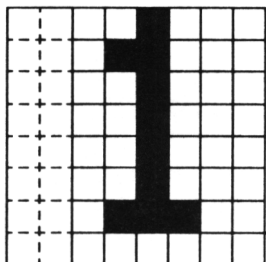
Our picture of the ORIC is not complete - there's no 'ORIC-1' logo in the top right-hand corner. To help you to put letters and numbers on to the HIRES screen, there is a BASIC command:

CHAR X,S,FB

where X is the ASCII code of the character we require (0-127), S is 0 or 1 depending upon which character set the character is to be taken from, and FB is 0,1,2 or 3.

To put the logo on to our picture, you must understand how the CHAR command operates.

You will recall from Chapter 11 that characters are made up of dots on an 8x6 grid. The number 1 would look like this:



To put a number 1 on the HIRES screen, we must position the cursor to a point corresponding to the top left-hand corner of the character grid. Type:

HIRES

then **CURSET 120,100,1**

The cursor will appear in the middle of the screen (we can see it because we used an FB number of 1 in the **CURSET** statement).

Now type

CHAR 49,0,1 (49 is ASCII for 1)

You will see that the 1 appears with the top left-hand corner of its character grid at the previously set cursor position.

The use of **CHAR** does not move the cursor. Try typing

CHAR 79,0,1

The letter O is superimposed on the 1.

With the **CHAR** command you can position characters on the **HIRES** screen with much greater precision than is possible in the other modes. For example, try this program:

```
10 HIRES
20 CURSET 0,100,1
30 FOR I = 1 TO 20
40 :   CURMOV I,2,0
50 :   CHAR 65,0,1
60 NEXT I
70 END
```

You might like to experiment along these lines - some spectacular effects can be achieved.

Getting back to our picture of the ORIC, you can incorporate the ORIC-1 logo using the **CHAR** command by adding these lines to the program:

```
1800 CURSET 190,54,0
1810 L$ = "ORIC-1"
1820 FOR J = 1 TO LEN(L$)
1830 :   CHAR ASC(MID$(L$,J,1)),0,1
1840 :   CURMOV 5 + (-1*(J=1)),0,0
1850 NEXT J
```

Each character of string L\$ is successively placed on the screen, with a space of 5 between each pair to squeeze the characters slightly closer together than is normal in **TEXT** mode. The spacing is controlled by the **CURMOV** statement in line 1840, which includes instructions to keep the O and the R six spaces apart.

Our picture is still not complete, because the ORIC is not all the same colour. We can 'colour in' blocks of a **HIRES** picture using the **FILL** command.

FILL R, C, N

FILL is used to 'paint' an area of picture, R rows by C character cells. The appearance of the area depends on the character, N, with which it is filled. N can be any number between 0 and 255.

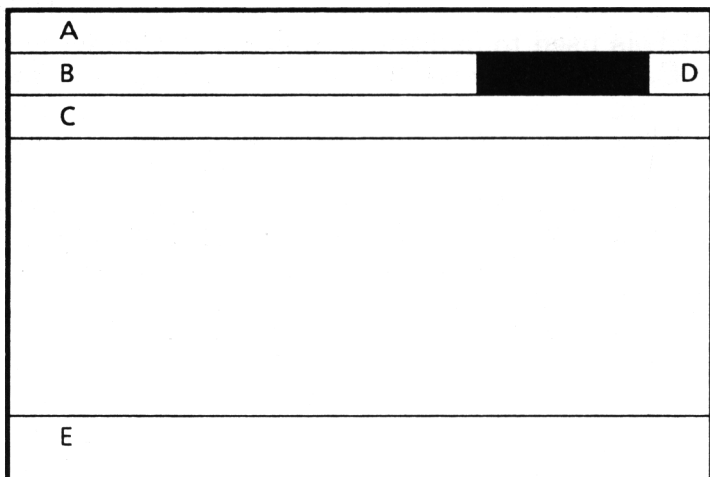
This program fills a square block in the middle of the screen (50 rows by 8 characters) with each of the possible values of N in turn, to allow you to see the effect. Values of N between 0 and 31 and between 128 and 159 are interpreted as attributes, so the program skips over those attributes which affect screen synchronisation.

```
10 HIRES
20 CURSET 100,75,0
30 FILL 50,8,N
40 GET A$
50 N = N+1
60 IF N=24 OR N=151 THEN N = N+8
70 PRINT N: GOTO 20
80 END
```

Returning to our ORIC picture program, we need to 'paint' the case white, except for the black area surrounding the logo. We can do this by adding five extra lines - the ORIC is divided into five rectangular sections, each of which must be filled separately.

To FILL these areas add the following lines to the program:

```
1900 CURSET 18,175,0:FILL 15,37,63
1910 CURSET 18,50,0: FILL 2,37,63
1920 CURSET 227,50,0:FILL 13,3,63
1930 CURSET 18,52,0: FILL 11,28,63
1940 CURSET 18,62,0: FILL 18,37,63
```



The five areas to be FILLed

One thing remains to complete our picture - the two blue lines above the keys. These can be added by placing the attribute for blue foreground in the character space to the left of the picture, corresponding to the position at which we want the lines.

```
1950 POKE 43801,4:POKE 43761,4
1960 POKE 43921,4:POKE 43961,4
```

Our picture is complete!

SAVING HIRES PICTURES ON TAPE

If you create a HIRES picture you would like to keep, you can save it on tape (and subsequently reload it) by following the instructions in Chapter 14.

DRAWING CIRCLES

So far we have only drawn straight lines in HIRES mode. The ORIC can also draw circles using the BASIC command **CIRCLE**.

CIRCLE R, FB

The two parameters in the **CIRCLE** command control the circle radius **R** and the **FB** number. The circle is drawn centred on the current cursor position, which is set to the centre of the screen in this example:

```
10  HIRES
20  CURSET 120,100,1
30  CIRCLE 50,1
```

Note that if you use a radius which would cause part of the circle to be outside the screen area the ORIC will respond with the message

?ILLEGAL QUANTITY ERROR

On many ORICs there is a bug which causes the circles to be drawn as ovals. There is no easy way to overcome this problem.

DOTTED CIRCLES

The **PATTERN** command can be used to produce dotted circles; try adding the line

```
15  PATTERN 1
```

to the last program.

Here is a program which uses **CIRCLE** and other **HIRES** commands to create a timer.

```
10  Z=4:R=80:X=120:Y=100
```

```

20  N=63
30  ST = 2*PI/60
40  HIRES
50  GOSUB 500
60  CLS:INPUT "DELAY TIME IN
    SECONDS";T:T1 = 0

100 C=PI
110 REPEAT
120 : CURSET X + R*SIN(C),
    Y+R*COS(C),1
130 : DRAW X-PEEK(537), Y-
    PEEK(538), 2
140 : WAIT N
150 : CURSET X + R*SIN(C),
    Y+R*COS(C),1
160 : DRAW X-PEEK(537), Y-
    PEEK(538), 2
170 : C = C-ST: T1 = T1 + 1
180 UNTIL T1 = T
190 REPEAT
200 : PING:WAIT N
210 UNTIL KEY$ = " "
220 GOTO 60

500 C = PI
510 CURSET 120,100,0: CIRCLE
    R+4,1
520 REPEAT
530 : CURSET X + R*SIN(C),
    Y+R*COS(C),1
540 : Z = Z+1
550 : IF Z/5 <> INT(Z/5) THEN 630
560 : DRAW X+(R - 5)*SIN(C) -
    PEEK(537), Y+(R - 5)*COS(C)
    -PEEK(538), 1
570 : CURSET X+(R+12)*SIN(C)-5,
    Y+(R+12)*COS(C), 0
580 : F$ = STR$(Z-5)
590 : CHAR ASC(MID$(F$,2,1)),0,1
600 : IF LEN(F$) = 2 THEN 630

```

```
610 : CURMOV 6,0,0
620 : CHAR ASC(MID$(F$,3,1)),0,1
630 : C = C-ST
640 UNTIL C < -PI
650 RETURN
```

Lines 10 to 60 set up the variables and input the time required. The value of N in line 20 controls the timing of the program, and you'll need to change this if you amend the program. Subroutine 500 draws and labels the timer clock face. Lines 100 to 180 draw and move the second hand, and lines 200 to 220 ring the bell until you press the space bar.

INK AND PAPER IN HIRES MODE

The global commands **INK** and **PAPER** have their usual effect in **HIRES** mode. **PAPER** controls the colour of the background while **INK** controls the colour of the foreground.

ATTRIBUTES IN HIRES MODE

The **HIRES** display can be controlled with attributes in the same way as **LORES** and **TEXT** displays. We used attributes to add the blue lines to the picture of the ORIC earlier in this chapter by **POKE**ing the code for blue foreground on to the screen, just to the left of the place where we needed the lines.

The point to remember is that in **HIRES** mode, for attributes, the screen is divided into 200 rows of 40 columns. An attribute can be placed in any one of these 8000 locations.

The following program illustrates this by putting a random background attribute into every location - it takes quite a while to run!


```
10 HIRES
20 FOR I = 40960 TO 48960
30 POKE I,12+RND(1)*8
40 NEXT I
```

In the same way we can control foreground colours.

```
10 HIRES
20 CURSET 120,100,0
30 FOR I = 1 TO 50
40 CIRCLE I,1
50 NEXT I
60 FOR I = 40960 TO 48960 STEP 40
70 POKE I,RND(1)*8
80 NEXT I
```

You can also make use of the flashing attributes in HIRES mode; try adding these lines to the last program.

```
90 FOR I = 40960 TO 48960 STEP 80
100 POKE I,13: NEXT
```

This will replace alternate foreground attributes down the left-hand side of the screen with flashing attributes.

As in other modes attributes take up a space on the screen which cannot be used to display anything else.

SUMMARY

In HIRES mode the screen is divided into 200 rows of 240 columns, with a three-line text window at the foot of the screen.

There are nine BASIC commands giving you control of the HIRES display.

HIRES selects high - resolution mode and places the **HIRES** cursor at the top left-hand corner of the screen (0,0).

DRAW X, Y, FB draws a line from the current cursor position to a position **X** across and **Y** down, using colours specified by the **FB** number.

CURMOV X, Y, FB moves the **HIRES** cursor to a position specified by the **relative** coordinates **X** and **Y** in a manner similar to the **DRAW** command.

CURSET X, Y, FB places the cursor at the position specified by **X** and **Y**, which in this case are **absolute** coordinates.

POINT (X,Y) returns a value of 0 if the specified point on the **HIRES** display is in background colour, and -1 if the point is in the foreground colour.

PATTERN N sets the pattern with which lines are drawn. **N** is a number between 0 and 255 and defines a binary repeating pattern for the line.

CHAR X,S,FB places the character whose ASCII is **X** on the screen at the current cursor position. **S** specifies the character set from which it is to be taken, and **FB** the colour in which it is to be drawn.

FILL R,C,N fills an area of the screen **R** rows by **C** columns with character **N**, starting at the current cursor position.

CIRCLE R draws a circle of radius **R** centred on the current cursor position.

The characteristics of the **HIRES** screen can be controlled using **INK** and **PAPER** commands and attributes, in the same way as the **TEXT** screen.

CHAPTER 14

LOADING AND SAVING PROGRAMS

When you switch off your ORIC, any programs and data you have typed in will be lost. Fortunately, it is possible to save the contents of the ORIC's memory on to a cassette tape using a standard domestic tape recorder, connected as described in Chapter 2.

In order to do this, the computer converts the data forming the program into audio signals, which are then recorded. When the program is later loaded, these audio signals are converted back into data.

Oric programs can be saved and loaded at two speeds, normal and slow. Programs must be loaded at the same speed as the one at which they were saved.

SAVING A PROGRAM

To save a program onto tape at normal speed, insert a blank tape into the recorder, making sure that you've wound the tape past the plastic leader. Type the following command :

```
CSAVE "Progame 1"
```

where 'Progame 1' can be any name you like, up to a maximum of 17 characters.

Now set the recorder to RECORD and press RETURN. Whilst the program is being saved, the message

Saving...Prognose 1

will appear on the top line of the screen. When the program has been saved, the 'Ready' prompt will appear on the screen, and the cursor will resume flashing.

LOADING A PROGRAM

To load a program from tape into the ORIC, insert the appropriate tape and type

CLOAD "Prognose 1"

Set the recorder to PLAY and press RETURN. The ORIC will search the tape until it finds the program called 'Prognose 1', and then load it. Whilst searching is taking place, the message

Searching...

appears on the top line of the screen. As soon as the program is found, the message will change to

Loading "Prognose 1"

When loading is complete, the cursor will resume flashing, and the 'Ready' prompt will appear on the screen.

You can instruct the ORIC to load the first program it finds on the tape by typing

CLOAD ""

Details of slow speed loading and saving are given below.

LOADING ERRORS

If the ORIC is unable to load your program, it will return to immediate mode, and display the message

FILE ERROR - LOAD ABORTED

Loading errors can occur for many reasons, but can be avoided by using good quality tapes (preferably special computer cassettes), ensuring that the Record / Play head on the recorder is clean (use a proprietary tape head cleaner / demagnetiser), and by experimenting with volume and tone controls until you find the settings most suitable for your recorder.

A major cause of problems when loading commercial software is tape head alignment. The Record/Play head in the tape recorder must be aligned so that it is parallel to the direction in which the tape runs. Most recorders have provision for adjustment by means of a small screw, located near the head. If this is the case, it's worthwhile trying adjustment whilst a music tape is running, adjusting the head until the clearest, sharpest sound is obtained. Having said that, the ORIC's cassette interface is very reliable, and it is unlikely that you will experience any problems.

SLOW SPEED LOADING AND SAVING

There is a method of ensuring that programs are saved more reliably; by saving them at a slower speed. That is, by instructing the ORIC to output the data at a much slower rate than normal, so that

you can be sure your program is correctly saved. To do this, type

```
CSAVE "Progame 1",S
```

and to load such a program

```
CLOAD"Progame 1",S
```

This means that it will take about eight times longer to load and save programs.

AUTO-RUN PROGRAMS

There is another technique you can use when saving programs, which will cause them to RUN immediately they are loaded. Simply type

```
CSAVE"Progame 1",AUTO
```

To load such a program, proceed in the usual way, and when loading is complete, the program will RUN automatically.

SAVING CHARACTER SETS AND PICTURES

If you have created a screen display or character set you wish to save on tape, you can do so by saving the block of memory containing the data as follows.

HIRES Picture

```
48K  CSAVE "Hires picture",  
      A#A000, E#BFE0
```

```
16K  CSAVE "Hires picture",  
      A#2000, E#3FE0
```

NOTE The ORIC must be in HIRES mode when loading or saving a HIRES picture.

LORES or TEXT Display

48K CSAVE "Lores Picture",
 A#BB80, E#BFE0

16K CSAVE "Lores picture",
 A#3B80, E#3FE0

The addresses you specify can be Hex or decimal so

48K CSAVE "Text", A48000, E49119

would work just as well.

Character Sets

To save the alternate character set, use:

48K CSAVE "Characters", A#B800,
 E#BB80

16K CSAVE "Characters", A#3800,
 E#3B80

In each of these cases, loading a memory block is performed in the usual way,

CLOAD "Characters"

You can load a memory block whilst you have a BASIC program stored in the computer, without disturbing it.

Finally, it is always advisable to keep backup copies of valuable programs, recorded at slow speed and clearly labelled. To prevent your accidentally

overwriting them, you can remove the two plastic tabs from the back edge of the cassette. Always handle cassettes carefully, and never put them near strong magnets or the T.V. for long periods, or you may erase the data.

CHAPTER 15

ADVANCED PROGRAMMING

The previous chapters cover all you need to write programs for your ORIC. This chapter describes some final BASIC commands which you will find useful as you write more ambitious programs, and rounds off with some general advice on how to write good programs.

TRACING BUGS IN PROGRAMS

It seems to be a fundamental law of nature that no computer program of any complexity will work first time. You will find that however carefully you plan and write a program, it will have one or two errors (usually called 'bugs' in the computer world) when you first type it in. To help you find bugs in programs the ORIC has a 'trace' facility, controlled by the two commands **TRON** and **TROFF**. This trace prints on the screen the program line number of each instruction executed by the ORIC as the program runs.

The trace controlling commands **TRON** and **TROFF** may only be used within programs. **TRON** switches on the trace facility; **TROFF** switches it off. To use the trace effectively, you should insert **TRON** and **TROFF** in the program before and after the section of program which you suspect to be faulty. The ORIC will then print the line numbers of all the instructions it executes between **TRON** and **TROFF**, allowing you to compare what actually happens with what should be happening.

STACKS, LOOPS AND SUBROUTINES

Two unusual commands, **POP** and **PULL**, are used in connection with subroutines and **REPEAT ... UNTIL** loops.

Suppose you have a program with subroutines nested one within another, one of which checks which keys are being pressed using **KEY\$**. You may wish to respond to a certain key - **ESC** perhaps - by abandoning the current routines and returning to the start of the program. This could mean **RETURN**ing through three or four levels of subroutine, which would be tedious to program, requiring **IF ... THEN RETURN** instructions at each level. The ORIC allows a simpler way of doing this, using the **POP** command.

Each time a **GOSUB** command is executed, the ORIC stores the location of the instructions after the **GOSUB** on a **stack** in the reserved area of memory. At any time, the stack contains return addresses for all the **GOSUB**s for which **RETURN** commands have not yet been performed. The **POP** command removes the most recently added **RETURN** address from the stack, so that the next **RETURN** sends the ORIC back to the **GOSUB** before that most recently added.

Try this example program.

```
10    TRON
20    GOSUB 100
30    PRINT "MAIN PROGRAM"
40    END

100   GOSUB 200
110   PRINT "SUB 100"
120   RETURN

200   GOSUB 300
```

```
210 PRINT "SUB 200"  
220 RETURN  
  
300 RETURN
```

If you run this, the ORIC will follow the **GOSUBs** to line 300 and then **RETURN** progressively through each subroutine, printing all the messages. The trace facility will print each line number as it is executed. If you now change line 300 to

```
300 POP: POP: RETURN
```

and **RUN** again, the program will only print 'MAIN PROGRAM', and the trace will reveal that the ORIC returned direct from line 300 to line 30.

Be careful when using **POP** - it can be very useful, but it can also make programs very hard to debug.

There is a similar command, **PULL**, which is used to remove the return addresses of **REPEAT ... UNTIL** loops from their stack. Each **REPEAT** command location is stored so the ORIC knows where to return to after **UNTIL**. This means that if you make the program repeatedly jump out of loops, the ORIC will eventually run out of storage for all the **REPEAT** addresses. This can be avoided by removing the last **UNTIL** return address from the stack with **PULL** when jumping out of the loop.

THE ORIC'S MEMORY

When you switch on the ORIC you see the message

```
47870 BYTES FREE.
```

This is a measure of the amount of storage space there is in the ORIC for programs and data.

The memory of a computer is arranged as a very large number of storage units, each of which can hold one number. Each of these storage units is called a **memory location**, and each location has a unique reference number called its **address**. In microcomputers like the ORIC each memory location can hold an 8-bit binary number which can have any value between 0 and 255. (Binary numbers are explained in Appendix 8.) Each location can be used to store one character of a program or string, but numbers need more than one location.

In computer jargon, eight bits of storage is called a **byte**. When the ORIC tells you it has 47870 bytes free, it means that 47870 storage locations are available for use. This is not, however, the number you can actually use, as the ORIC reserves a portion of the memory for its own purposes. (All these figures are for the 48K Version 1.0 ORIC; the 16K model and Version 1.1 models will give different numbers.)

What does the ORIC do with this memory? About 1300 bytes are reserved as storage areas for data. Another 9000 bytes or so are set aside to store the display picture and the character sets for the display. This leaves around 39500 bytes for you to use for BASIC programs, and as variable storage for these programs. 39500 bytes is about $38\frac{1}{2}$ K (1K in computerese is 1024 bytes - 2^{10}), which may seem like a colossal amount. However a program which uses a lot of variable storage - particularly one storing many strings and arrays, will use up this store in large quantities. There is a diagram of the ORIC memory in Appendix 7.

There are a few BASIC commands to help you manage this memory and so squeeze the most out of your ORIC.

The function **FRE(0)** will give the amount of store left unused by BASIC. If you **PRINT FRE(0)** immediately after switching on you ORIC it will print 39421 (for the 48K machine). Enter a program, and **PRINT FRE(0)** again, and a smaller number will be printed. If you **RUN** the program and **PRINT FRE(0)** a third time the number printed will be smaller still, as the ORIC has taken memory space to store the variables.

FRE has a second purpose: it forces the ORIC to perform a **garbage collection**. While a program is running, variables are constantly being redefined, and the ORIC does not always re-use the same memory when string variables are reset. A command such as

```
340 PRINT FRE("") or
```

```
340 FF = FRE("")
```

forces the ORIC to sort out and repack its variables storage area to make more efficient use of the space. The use of **FRE("")** in programs that use nearly all the memory will make them less likely to crash with **OUT OF MEMORY** errors. However, the garbage collection procedure takes time to operate, so don't put **FRE("")** instructions inside loops or your programs will be slowed down.

The **CLEAR** command may occasionally be useful. The command clears all the variables stored by a program, and should therefore be used with extreme care, but if you have a program which is divided into a number of almost independent sections you may be able to save memory by **CLEARing** before beginning each section.

Remember that **CLEAR** deletes all variables: if you have some data which must be passed from one section of the program to another you can use the commands **POKE** and **DOKE** to store the data directly in the memory, to be read into variables again by the next program segment, using **PEEK** or **DEEK**. (These commands are described later in this chapter.)

If you do run out of memory with a large program, it is possible to gain another 8000 bytes of memory with the command **GRAB**. The **HIRES** display uses four times as much store as the **TEXT** and **LORES** displays, and the memory reserved for this display is not used when the **ORIC** is in **TEXT** or **LORES** modes. If you use the **GRAB** command, the unused memory is freed for use by **BASIC** programs. You may not use the **HIRES** display after a **GRAB** command, as the storage area is no longer available for the display. To restore the normal memory arrangement of the computer use **RELEASE**, which restores the **HIRES** screen, and takes back the 8000 bytes of memory from **BASIC**.

MAKING USE OF MEMORY

There are four commands, **PEEK**, **POKE**, **DEEK** and **DOKE**, which allow you to make direct use of the **ORIC**'s memory.

PEEK allows you to find out the contents of a memory location:

```
PRINT PEEK(623)
```

prints the contents of location 623. (The result is usually 27, as the **ORIC** uses this location to store the number of lines in the **TEXT** and **LORES** screen displays).

POKE puts numbers into storage locations:

POKE 623, 12

puts 12 into location 623. (This will have the effect of reducing the number of lines you can use on the LORES and TEXT displays.)

DEEK reads the contents of two locations at once. The two locations are considered to be the low and high parts of a 16-bit binary number, with the low byte in the lower numbered location.

PRINT DEEK(621)

prints the 16-bit number stored in 621 and 622. In other words, the number printed is the contents of location 621 added to 256 times the contents of location 622, i.e.

DEEK (621)

is equivalent to

PEEK(621) + 256*PEEK(622)

The contents of 621 and 622 will usually be 48000, as these locations are used to store the start address in memory of the screen.

DOKE is the complement of **DEEK**, and stores an integer from 0 to 65535 in two consecutive memory locations. The number is stored as a 16-bit binary number with the lower eight bits in the lower numbered store location.

DOKE 621, 48400

stores 48400 in 621 and 622. Because these particular locations hold the start address of the screen, this command has the effect of moving the

'top' of the screen 10 lines down the display. As with **DEEK**, **DOKE** is equivalent to two **POKEs**:

```
DOKE 621,48400
```

is equivalent to

```
POKE 622, INT(48400/256): POKE621,  
48400 - PEEK(622)*256
```

NOTE The store locations given apply only to machines with the Version 1.0 operating system.

MACHINE CODE SUBROUTINES

The ORIC's microprocessor does not understand BASIC programs without some assistance; it understands a much cruder set of instructions called **machine code** or **machine language**. BASIC programs can be run only because the ORIC has some machine code programs permanently stored in read-only memory (ROM) which interpret the BASIC. Writing programs in machine code is more difficult than using BASIC, but can be very worthwhile as machine code programs run very much faster than BASIC. Machine code is beyond the scope of this book, but there are several books available about the ORIC's microprocessor (a 6502) and its instruction code as it is very common in personal computers.

You can enter machine code routines from BASIC using the command **CALL**, giving the start address in memory of the machine code routine. For example

```
CALL 555
```


calls the routine which is normally triggered by the reset button underneath the ORIC, so you can reset the machine without turning it over.

CALL 62509

will call the power-on setting up routines which clear the memory and print the switch-on message. **BE CAREFUL** using this - it will destroy any program you have in the ORIC at the time you call it.

Machine code routines may also be called by the instruction **USR()**. The instruction is used like a BASIC function. For example:

1000 A = USR(X)

The number in brackets is stored in the ORIC's floating point accumulator, and then a previously defined machine code routine is called. At the end of the routine, the number now in the floating point accumulator is returned to be stored in the variable.

The address of the machine code to be run is defined by **DEF USR**, like this:

DEF USR = #0400

would cause subsequent **USR()** instructions to run the code starting at address #0400 (=1024).

Setting the Limits of Memory

To reserve space in the memory for machine code programs or for storage you can set the limit of the store available to BASIC by using the **HIMEM** command. The highest location which BASIC may use is normally 40704, unless **GRAB** has been used

in which case it is 47871 (the numbers are different for the 16k ORIC). The value is stored in locations 166 and 167, and can be inspected using DEEK(166). To set the top of memory, use HIMEM. For example

```
HIMEM 39704
```

will reserve 1000 bytes for machine code use, and these bytes will not be usable for BASIC program or variable storage. The top of memory is reset by the reset button (or by CALL 555), so be careful!

DESIGNING GOOD PROGRAMS

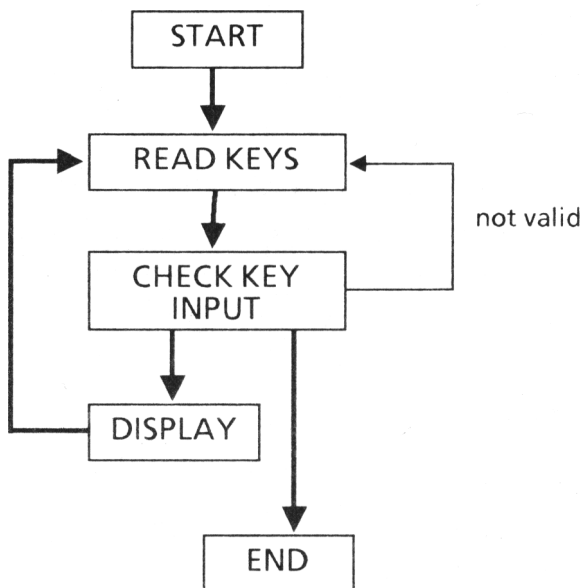
It is easy to write simple BASIC programs, and after some practice you will find it fairly easy to write quite complicated routines. However, unless you plan your programs carefully, a long program can get very messy and it can be very difficult to find all the bugs.

To write complex programs successfully you must design them carefully, following a few simple rules. The phrase **Structured Programming** is often used to describe these rules ; all this means is that programs designed in accordance with these rules have a clear and logical structure, and the flow through the program is easy to follow. You may be put off by hearing people say that BASIC is not a suitable language for structured programming. This isn't true. While BASIC may lack the elegance of some other languages, it is still possible to use it to write good programs by following the simple guidelines set out below.

- 1 Always plan out the program on paper. Decide what you want the program to do, identifying the various tasks it will have to perform, and divide the program into sections

which correspond to the tasks. For example, you might have one section of program to get data from the keyboard, another to sort the data, a third to display results, and so on.

- 2 It is often useful to draw sketches of the way the tasks fit together.



- 3 When you have got the overall structure of the program sorted out, work out in more detail what each section should be doing, and how it should be done.
- 4 Write the BASIC routines for each section of the program - writing them on paper and **not** typing straight into the ORIC. Try to write each section of the program so that it is as independent as possible of the other sections,

and arrange the programs with a clear sequential flow from beginning to end: avoid jumping around with **GOTO**. Use a different set of line numbers for each section, beginning each at a round number of so many thousands or tens of thousands.

- 5 If using **GOSUB**, use a name for the first line number of the subroutine - see the examples in the next chapter. This makes programs much easier to follow.
- 6 Now type the program into the ORIC.
- 7 The program will not work! At this stage you can sort out all the SYNTAX ERRORS easily. If you have designed the program well, with different sections doing different tasks, it will be much easier to find the more subtle errors, as you will be able to isolate the faulty bit of the program without difficulty.

If you follow these guidelines, you will save a lot of time in getting your programs to work, and you will have more time for actually using the programs.

CHAPTER 16

EXAMPLE PROGRAMS

The programs in this chapter are included both to show you the potential of the ORIC, and as examples of the way in which you should plan and write your programs to make them efficient and easy for you to understand. We have deliberately kept the programs fairly short, so that you will not have to spend too long typing them in, and they all have possibilities for further development which we leave up to you.

SKETCHPAD

This simple program allows you to make line drawings on the HIRES screen by using the cursor control keys to move a 'pen' on the display. There are three modes of operation :

DRAW	draws lines
MOVE	moves cursor
ERASE	erases lines

The movements are controlled by the four cursor keys. Use the ESC key to halt the program.

```
10      REM    SKETCHPAD
20      REM
90      REM THE FIRST PART SETS UP
        THE DISPLAY AND THE
        VARIABLES
100     HIRES
```

```
110 CX = 120: CY = 100
120 CURSET CX, CY, 0 'PUT CURSOR
    IN SCREEN
    CENTRE

140 POKE 618, 10
160 CLS 'CLEAR TEXT LINES
170 PRINT: PRINT "DRAW
    ERASE MOVE ESCAPE"

190 REM SET UP CONSTANTS
200 L$ = CHR$(8) 'CURSOR KEY
    CHARS
220 R$ = CHR$(9)
230 UP$ = CHR$(11)
240 DOWN$ = CHR$(10)
250 ESC$ = CHR$(27) 'ESC KEY
260 HOME$ = CHR$(30) 'MOVES
    CURSOR TO TOP LEFT
270 LBLANK$ = CHR$(14) 'BLANKS
    LINE

290 REM SUBROUTINE ADDRESSES
300 CFLASH = 2000 'CURSOR FLASH
310 CDRW = 10000 'DRAWS THE
    CURSOR
320 LEFT = 3000 'DRAWS/MOVES
    LEFT
330 RIGHT = 4000 'RIGHT
340 UP = 5000 'UP
350 DOWN = 6000 'DOWN
360 MESSAGE = 7000 'ALTERS
    TEXT
    MESSAGE
370 DISPLY = 11000 'DRAWS,
    ERASES OR
    MOVES

490 REM SET UP FOR "DRAW" MODE
500 ULINE$ = "----" 'UNDERLINE
```

```
510      FB = 1: C = 0
520      GOSUB MESSAGE

990      REM    MAIN PROGRAM
995      REM
1000     REPEAT
1010     WAIT 7 'SYNCHRO DELAY

1090     REM    READ KEYBOARD AND
          FLASH CURSOR
1100     :      REPEAT
1110     :          K$ = KEY$
1120     :          IF K$ = "" THEN GOSUB
          CFLASH 'FLASH CURSOR
1130     :      UNTIL K$ <> ""

1190     REM    CHECK KEYSTROKE
1200     IF K$ = L$ THEN GOSUB LEFT:
          GOTO 1300
1210     IF K$ = R$ THEN GOSUB RIGHT:
          GOTO 1300
1220     IF K$ = UP$ THEN GOSUB UP:
          GOTO 1300
1230     IF K$ = DOWN$ THEN GOSUB
          DOWN: GOTO 1300
1240     REM    CHECK FOR MODE CHANGE
1250     IF K$ = "D" THEN FB = 1:
          GOSUB MESSAGE
1260     IF K$ = "E" THEN FB = 0:
          GOSUB MESSAGE
1270     IF K$ = "M" THEN FB = 3:
          GOSUB MESSAGE
1300     UNTIL K$ = ESC$      'PROG
          ENDED BY ESC KEY

1390     REM    END OF PROG
1400     IF C = 1 THEN GOSUB CDRW
          'TURN OFF HIRES "CURSOR"
1410     CLS
```

```
1420  POKE 618, 3  'TURN ON CURSOR
      AND CLICK
1500  END

1900  REM  SUBROUTINES

1990  REM SUBROUTINE TO FLASH
      CURSOR
2000  GOSUB CDRW          'DRAW/ERASE
      CURSOR
2010  WAIT 30
2020  RETURN

2990  REM SUBROUTINE TO DRAW TO
      LEFT
3000  X = -1: Y = 0  'SET
                        DIRECTION OF
                        MOVE
3010  IF CX > 15 THEN GOSUB DISPLY
      'ONLY MOVE IF SPACE TO LEFT
3020  RETURN

3990  REM SUBROUTINE RIGHT
4000  X = 1: Y = 0
4010  IF CX < 234 THEN GOSUB
      DISPLY
4020  RETURN

4990  REM SUBROUTINE UP
5000  X = 0: Y = -1
5010  IF CY > 5 THEN GOSUB DISPLY
5020  RETURN

5990  REM SUBROUTINE DOWN
6000  X = 0: Y = 1
6010  IF CY < 194 THEN GOSUB
      DISPLY
6020  RETURN

6990  REM SUBROUTINE MESSAGE
```



```

7000  PRINT HOME$
7010  PRINT
7020  SP = -10*(FB = 0)-20*(FB =
      3)
7030  PRINT LBLANK$ SPC(SP)
      ULINE$;
7040  RETURN

9990  REM  SUBROUTINE CDRW
9995  REM  DRAWS/ERASES CURSOR

10000 DRAW -3, 3, 2
10010 CURMOV 7, 1, 3
10020 DRAW -7, -7, 2
10030 CURMOV 6, 0, 2
10040 DRAW -3, 3, 2
10050 C = (C = 0) 'C INDICATES
      CURSOR ON/OFF
10060 RETURN

10990 REM  SUBROUTINE DISPLY
10995 REM  DRAWS/MOVES/ERASES
      UNDER CONTROL
10997 REM  OF LEFT, RIGHT, UP AND
      DOWN

11000 IF C THEN GOSUB CDRW
      'ERASE CURSOR
11010 DRAW X, Y, FB
11020 CX = CX + X      'NEW CURSOR
      POSITION
11030 CY = CY + Y
11040 RETURN

```

PROGRAM DESCRIPTION

The program divides into five sections . Lines 10 - 520 prepare the ORIC and define the variables, lines 1000 - 15000 are the main drawing program,

and the remainder of the program consists of the subroutines called by the main program.

Preparation Lines 100 - 520

Lines 100 to 170 select HIREs mode, position the cursor in the screen centre and print the message at the foot of the screen. The **POKE** command in line 40 disables the TEXT cursor and the key-click.

Lines 200 to 370 define the variables which remain constant throughout the program: 200-260 defining characters, 300-370 defining subroutine addresses. The use of variables to store subroutine addresses makes the main program easier to understand, and also means that in the event of the program being renumbered, only one instruction defining the subroutine line number will need to be changed.

Main Program Lines 1000 - 15000

The drawing routines are all contained within the **REPEAT ... UNTIL** loop stretching from 1000 to 1300. The ORIC cycles round this loop continually until the ESC key is pressed to end the program.

An inner loop at 1100 - 1130 repeatedly checks the keyboard and flashes the cursor until a key is pressed. The **WAIT** at line 1010 synchronises the execution time of the loop with the key repeat time, so that if you hold down a key the line will be drawn smoothly without pausing to flash the cursor.

The lines 1200 - 1230 check for a cursor keypress and call the appropriate subroutine.

Lines 1250 - 1270 alter the drawing mode as required. The drawing mode (DRAW, ERASE or MOVE) is controlled by setting the variable FB to 1

(DRAW), 0 (ERASE) or 3 (MOVE). The variable FB is then used as the FB parameter of a DRAW command in subroutine DISPLY.

The lines from 1400 - 1500 re-enable the TEXT cursor and the keyclick before ENDing the program.

Subroutines

CFLASH (2000) calls CDRW to draw or erase the cursor and pauses before returning.

LEFT (3000), RIGHT (4000), UP (5000) and DOWN (6000) set the direction in which the cursor is to move, check that it is not too close to the edge of the screen and call DISPLY to move the cursor.

MESSAGE (7000) underlines the chosen mode at the foot of the screen.

CDRW (10000) draws or erases the cross-hair cursor. The FB parameter used in the DRAW comands is 2, which reverses whatever is on the screen at the time, so the cursor is alternately drawn and erased. The variable C is switched between 0 and -1 (FALSE and TRUE - see Chapter 9) to indicate the current cursor state.

DISPLY (11000) erases the cursor if C is TRUE (cursor ON), and the DRAW instruction in line 11010 draws or erases a line or moves the cursor as required. The new cursor position is stored in CX and CY.

SYNTHESISER

This program turns your ORIC into an electronic organ, with the top two rows of the keyboard becoming the organ keyboard. Four different

'voices' are provided: Bass, Clarinet, Mandolin and Vibrato.

When you **RUN** the program a picture of the keyboard is drawn, showing which ORIC key plays which note. The four voices are selected by pressing the appropriate keys: B for Bass, C for Clarinet, and so on. Pressing the space bar ends the program.

```
10    GOSUB 1000
20    GOSUB BEGIN
30    GOSUB TUNE
40    GOSUB FINISH
50    END

990   REM    SETTING UP ROUTINES
995   REM    FIRST READ THE KEYBOARD
      DATA

1000  CLS
1010  DIM NO%(127,1)      ' NO% HOLDS
      THE NOTES FOR EACH KEY
1020  REPEAT
1030  : READ K, NO%(K,0), NO%(K,1)
1040  UNTIL K=127

1090  REM    SET SUBROUTINE ADDRESSES
1100  BEGIN = 2000 'DRAWS THE
      KEYBOARD AND SETS MANDOLIN
1110  TUNE = 3000    'THE MAIN
      PLAYING ROUTINE
1120  NO = 4000      'PLAYS THE
      NOTES
1130  INSTRUMENT = 5000 'CHANGES
      THE VOICES
1140  MESSAGE = 6000   'DISPLAYS
      CHOSEN VOICE
1150  FINISH = 7000    'ENDS
      PROGRAM
```

```

1190 REM    DISABLE CURSOR AND
        CLICK
1200 POKE 618, 10
1500 RETURN

1990 REM    SUBROUTINE BEGIN
1995 REM    DRAWS KEYBOARD AND
        SELECTS FIRST VOICE
2000 CLS
2010 B$ = CHR$(160)          'BLACK
        SQUARE
2020 K2$ = " | "+B$+" "+B$+" "
2030 K3$ = K2$ + B$+" "
2040 L$ = " | "+K2$ + K3$ + K2$ +
        K2$ + " |"
2050 PLOT 2,1, CHR$(9) +
        "@PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
        PPPP0"
2060 PLOT 3, 2, L$
2070 PLOT 3, 3, L$
2080 PLOT 3, 4, L$
2090 PLOT 9, 3, CHR$(178) + " " +
        CHR$(179)
2100 PLOT 15, 3, CHR$(181) + " " +
        CHR$(182) + " " + CHR$(183)
2110 PLOT 23, 3, CHR$(185) + " " +
        CHR$(176)
2120 PLOT 29, 3, CHR$(189) + " " +
        CHR$(220)
2130 PLOT 3, 5, " | | | | | | | |
        | | | | | | |"
2140 PLOT 3, 6,
        "|ESC|Q|W|E|R|T|Y|U|I|O|P|[|]
        |DEL|"
2150 PLOT 2, 7, CHR$(9) + CHR$(34)
        + "#####
        #####!"

2200 DOKE 621, 48320 'LOWER TOP OF
        SCREEN

```

```
2210 POKE 623, 19      'REDUCE SCREEN
                        HEIGHT

2300 K$ = "M"          'SET
                        "MANDOLIN"

2310 GOSUB INSTRUMENT
2320 GOSUB MESSAGE

2500 RETURN

2990 REM SUBROUTINE TUNE
2995 REM WHERE IT ALL HAPPENS

3000 REPEAT
3010 : REPEAT
3020 :   K$ = KEY$ 'READ THE KEYS
3030 :   UNTIL K$ <> ""
3040 :   REM CHECK THE KEYSTROKE
3050 :   N = NO%(ASC(K$),1)
3060 :   IF N=0 THEN GOSUB
                        INSTRUMENT: GOSUB MESSAGE:
                        GOTO 3100
3070 :   O = OC + NO%(ASC(K$),0)
3080 :   GOSUB NO
3100 UNTIL K$ = " "
3200 RETURN

3990 REM SUBROUTINE NO
3993 REM SOUNDS THE NOTES
3995 REM
4000 MUSIC 1, O, N, O
4010 MUSIC 2, O, N, O
4020 PLAY CH, Z, E, P
4100 RETURN

4990 REM SUBROUTINE INSTRUMENT
4995 REM CONTROLS THE VOICE
4998 REM
```

```
5000 IF K$="B" THEN E=1: P=1750:
      Z=0: CH=3: OC=1: I$=K$
5010 IF K$="C" THEN E=1: P=5000:
      Z=0: CH=1: OC=2: I$=K$
5020 IF K$="M" THEN E=1: P=2000:
      Z=0: CH=3: OC=3: I$=K$
5030 IF K$="V" THEN E=4: P=60 :
      Z=1: CH=3: OC=3: I$=K$
5100 RETURN
```

```
5990 REM  SUBROUTINE MESSAGE
5994 REM  DISPLAYS VOICE OPTIONS
5998 REM
6000 RED$=" " + CHR$(27)+"A"
6010 B$ = "  BASS"
6020 C$ = "  CLARINET"
6030 M$ = "  MANDOLIN"
6040 V$ = "  VIBRATO"
6050 IF I$ = "B" THEN
      B$=RED$+MID$(B$,3)
6060 IF I$ = "C" THEN
      C$=RED$+MID$(C$,3)
6070 IF I$ = "M" THEN
      M$=RED$+MID$(M$,3)
6080 IF I$ = "V" THEN
      V$=RED$+MID$(V$,3)
6100 PRINT CHR$(30) 'MOVE TO TOP
      OF SCREEN
6110 PRINT: PRINT: PRINT B$
6120 PRINT: PRINT: PRINT C$
6130 PRINT: PRINT: PRINT M$
6140 PRINT: PRINT: PRINT V$
6150 PRINT: PRINT: PRINT "PRESS
      SPACE BAR TO END"
6500 RETURN
```

```
6990 REM  RESET BEFORE ENDING
7000 PLAY 0,0,0,0
```

```
7010 POKE 618, 3      'TURN ON CLICK
      & CURSOR
7020 DOKE 621,48000 'RESET SCREEN
7030 POKE 623,27
7040 CLS
```

```
7500 RETURN
```

```
9990 REM    THE DATA
9992 REM    FOR ARRAY NO%
9994 REM
10000 DATA 27, -1, 12, 48, 1, 4,
      50, 0, 2
10010 DATA 51, 0, 4, 53, 0, 7,
      54, 0, 9
10020 DATA 55, 0, 11, 57, 1, 2,
      61, 1, 7
10030 DATA 69, 0, 5, 73, 1, 1,
      79, 1, 3
10040 DATA 80, 1, 5, 81, 0, 1,
      82, 0, 6
10050 DATA 84, 0, 8, 85, 0, 12,
      87, 0, 3
10060 DATA 89, 0, 10, 91, 1, 6,
      92, 1, 9
10070 DATA 93, 1, 8, 127, 1, 10
```

PROGRAM DESCRIPTION

The program is built of subroutines, with a very short main program calling the routines in order. The two-dimensional array NO% is used to hold the information relating the keyboard to the musical notes. Only those elements of the array corresponding to the ASCII codes of the keys are used; the others remain blank. This method is used to make the program faster, avoiding the use of many IF ... THEN instructions. The table shows how NO% is used.

NOTE	KEY	ASCII of Key K	NO%(K,0) Octave	NO%(K,1) Note
B	ESC	27	-1	12
C	Q	81	0	1
C#	2	50	0	2
D	W	87	0	3
D#	3	51	0	4
E	E	69	0	5
F	R	82	0	6
F#	5	53	0	7
G	T	84	0	8
G#	6	54	0	9
A	Y	89	0	10
A#	7	55	0	11
B	U	85	0	12
C	I	73	1	1
C#	9	57	1	2
D	O	79	1	3
D#	0	48	1	4
E	P	80	1	5
F	[91	1	6
F#	=	61	1	7
G]	93	1	8
G#	\	92	1	9
A	DEL	127	1	10

Subroutines

Subroutine 1000 reads the data into NO%, sets variables to hold the subroutine line numbers and switches off the flashing cursor and the keyclick by POKEing 10 into location 618.

Subroutine BEGIN (2000) uses low resolution graphics symbols to draw a keyboard. The character CHR\$(160) is a reversed space - a black square - and is used for the black keys. Characters CHR\$(176) to CHR\$(220) are reversed numbers and signs (white on black) and are used to label the black keys. The top of the screen is moved down to below the keyboard by lines 2200 and 2210 to protect it from scrolling off the screen. These POKEs are explained in Chapter 15.

Subroutine TUNE (3000) reads the keyboard and takes the appropriate actions. Two nested REPEAT ... UNTIL loops are used.

Subroutine NO (4000) sounds the notes.

Subroutine INSTRUMENT (5000) alters the voice by setting the envelope, period and channels used by the PLAY command; and also OC, which is used to select the octave used in the MUSIC commands.

Subroutine MESSAGE (6000) displays the instruments, with the chosen instrument highlighted in red.

CODEBREAKER

This program simulates the popular game in which your opponent (in this case the ORIC) picks a code in the form of four coloured pegs, and you have ten guesses to crack it. After each attempt, you are

given a clue in the form of a number of 1s and 0s. A 1 indicates a correct colour in the right place, and a 0 indicates a correct colour in the wrong place.

```
1      REM
2      REM CODEBREAKER
3      REM
5      CLS:PRINT CHR$(17) ' TURN OFF
      CURSOR
10     PRINTSPC(10)CHR$(4)
      CHR$(27)"J CODEBREAKER"
      CHR$(4)
20     PAPER2:INK0
27     REM
28     REM DISPLAY INSTRUCTIONS
29     REM
30     PRINT:PRINT:PRINT:PRINT "THE
      ORIC WILL SELECT A CODE OF
      FOUR"
35     PRINT "COLOURS, CHOSEN FROM
      SIX. EACH COLOUR"
40     PRINT "MAY BE USED MORE THAN
      ONCE":PRINT
45     PRINT "YOU MUST CRACK THE
      CODE IN LESS THAN"
50     PRINT "10 ATTEMPTS":PRINT
55     PRINT "AFTER EACH ATTEMPT YOU
      WILL BE GIVEN"
60     PRINT "A CLUE OF UP TO 4
      NUMBERS :":PRINT
65     PRINT "1 MEANS A CORRECT
      COLOUR IN THE RIGHT"
70     PRINT "PLACE":PRINT
75     PRINT "0 MEANS A CORRECT
      COLOUR IN THE WRONG PLACE"
76     PRINT:PRINT
80     PRINT:PRINT SPC(7) CHR$(140)
      "PRESS ANY KEY TO START"
85     GET A$
90     CLS:PAPER0:INK7
```

```
97 REM
98 REM CONTROL ROUTINE
99 REM
100 GOSUB 1000 ' SET CODE
110 FOR GUESS=1 TO 10:G$="":
    CODE$=C$
120 GOSUB 2000 ' INPUT GUESS
130 NEXT
140 Z$=CHR$(12)+"
    YOU LOSE":GOTO 2410

997 REM
998 REM SET CODE
999 REM
1000 C$=""
1010 FOR I=1 TO 4
1020 C$=C$+RIGHT$(STR$(INT(RND(1)
    *6)+1),1)
1030 NEXT I
1040 RETURN ' CODE AS STRING C$

1997 REM
1998 REM INPUT GUESS
1999 REM
2000 GN$="INPUT GUESS "+
    RIGHT$(STR$(GUESS),1)
2010 PLOT 7,0,GN$ ' GUESS NUMBER
2020 FOR I=1 TO 4
2030 GET I$
2040 IF I$ < "1" OR I$ > "6" THEN
    PING:GOTO 2030
2050 PLOT 20+I,2*GUESS+1,CHR$(16)
2060 PLOT 30+I,2*GUESS+1,I$
2070 PLOT 19+I,2*GUESS+1,
    CHR$(VAL(I$)+16)
2080 G$=G$+I$
2090 NEXT I

2097 REM
2098 REM RIGHT COLOUR/RIGHT PLACE
2099 REM
```

```
2100 FOR I=1 TO 4
2110 IF MID$(G$,I,1) < > MID$
    (CODE$,I,1) THEN 2150
2120 C1=C1+1
2130 G$=LEFT$(G$,I-1)+ "9"+
    MID$(G$,I+1)
2140 CODE$=LEFT$(CODE$,I-1)+"9"+
    MID$(CODE$,I+1)
2150 NEXT I

2197 REM
2198 REM RIGHT COLOUR/WRONG PLACE
2199 REM
2200 FOR I=1 TO 4
2210 FOR J=1 TO 4
2220 IF MID$(G$,I,1)=MID$(CODE$
    ,J,1)ANDMID$(G$,I,1)<>"9"THEN
    C2=C2+1 ELSE 2260
2240 G$=LEFT$(G$,I-1)+"9"
    +MID$(G$,I+1)
2250 CODE$=LEFT$(CODE$,J-1)+"9"+
    MID$(CODE$,J+1)
2260 NEXT J
2270 NEXT I

2297 REM
2298 REM DISPLAY CLUE
2299 REM
2300 IF C1=0 THEN 2320
2310 FOR I=1 TO C1:
    CLUE$=CLUE$+"1":NEXT
2320 IF C2=0 THEN 2340
2330 FOR I=1 TO C2:
    CLUE$=CLUE$+"0":NEXT
2340 PLOT 12,2*GUESS+1,CLUE$
2350 IF CLUE$="1111" THEN 2400
2360 C1=0:C2=0:CLUE$="":RETURN

2397 REM
2398 REM REVEAL CODE
2399 REM
```

```
2400 POP:Z$=CHR$(12)+"  
      YOU WIN !!!"  
2410 PLOT 0,0,Z$  
2420 PLOT 0,25,"THE CODE WAS"  
2430 FOR I = 1 TO 4  
2440 PLOT 13+I,25,CHR$(VAL(MID$(  
      C$,I,1))+16):NEXT  
2450 PLOT 18,25,CHR$(16)  
2460 PLOT 20,25,"PRESS Y FOR  
      REPEAT":GET A$  
2470 IF A$="Y" THEN CLEAR:GOTO 90  
2480 CALL 555 ' WARM RESET
```

PROGRAM DESCRIPTION

The program is divided into eight sections, each of which performs a separate task.

- 2-90 Displays the instructions and set up the display.
- 100-140 The controlling section of the program which calls the various subroutines for each guess.
- 1000-1040 Creates a string, C\$, of four random numbers between 1 and 6, which form the code you have to crack.
- 2000-2090 Inputs your guess, checks it and displays it on the screen both as a number and as a colour code made up of foreground attribute characters.
- 2100-2150 Checks for correct colour in the right place. For every such item C1 is incremented and that item in the guess, and in the code is set to '9', as a marker, so that it isn't considered

when checking for right colour in the wrong place.

2200-2270 Checks for right colours in the wrong place. Every such item in the guess and the code is set to '9', so that they aren't considered more than once, and the counter C2 is incremented.

2300-2360 Builds up a string, CLUE\$, containing a 1 for every correct colour in the right place - indicated by C1, and a 0 for every correct colour in the wrong place - indicated by C2. Note that there is no information in the clue as to the positioning of the correct guesses - that would make things too easy!

2400-2480 If you've cracked the code or run out of guesses, this routine lets you know and reveals the code. You then have the opportunity to play again, or quit.

RENUMBER

This program enables you to alter the line numbers of your programs and the increment between them. This is often a useful thing to be able to do, since it allows you to create space between lines for inserting extra code. To use the program, it must be loaded at the start of a programming session. You can then type in your program, and whenever you wish to renumber, activate the renumber program by typing RUN 60000. Obviously your own program should not use line numbers greater than 59999!

You can save the entire contents of memory at the end of the session, only deleting the renumber program when your own program is complete.

```
60000 CLS
60010 INPUT "START";S
60020 INPUT "STEP";ST
60030 AD=1281
60040 REPEAT
60050 : NL=DEEK(AD)
60060 : DOKE AD+2,S
60070 : S=S+ST
60080 : AD=NL
60090 UNTIL DEEK(AD+2)=60000
60100 END
```

In order to understand how the program works, let's look at how the ORIC stores a BASIC program. All BASIC programs start in memory at location 1280. This location is always set to 0. The next two locations contain a pointer to the address where the next line number is stored. The following locations contain the line number and the BASIC program line, which ends in a 0. Next comes a pointer to the next line number, and so on to the end of the program. A diagram will help to clarify the situation.

ADDR Contents

```
1280 Always 0
1281 2- byte pointer
1282 to next line number
1283 2- byte line
1284 number
1285
1286
1287
1288
```


1289 Program line
1290
1291
1292
1293
1294 0 indicates end of program line
1295 2- byte pointer
1296 to next line number
1297 2- byte line
1298 number
1299 Program
1300 Program etc. etc.

PROGRAM DESCRIPTION

60010,60020 Set up the new first line number and the increment between lines.

60030 Sets AD to the start of the program.

60040-60090 Sets NL to the start of the next line, changes the current line number to S and then increments S until the end of the program is reached.

THE BASIC TOKEN SYSTEM

In order to save memory space and increase running speed, the ORIC stores BASIC commands in a shorthand form known as Tokens. Instead of using a memory location for each character in a BASIC command, it uses a 1-byte code for each command.

For example, take the line :

10 GOTO 100

The line would be stored as described above, with a pointer to the next line number, a 2-byte line number and then the program line itself. Instead of using four bytes to store the characters

G O T O, the ORIC stores the number 151, which it recognises as **GOTO** when the program is **RUN**. The line number following the token is stored in its ASCII form - that is in three consecutive memory locations containing 49, 48, 48 respectively.

To see this type in the line, then type

```
FOR I = 1280 TO 1290 : PRINT  
PEEK(I) : NEXT
```

You could use this knowledge of the way in which BASIC is stored to improve upon the renumber program, making it cater for **GOSUB**, **GOTO**, **THEN** and **ELSE** commands. Should you like to try this, the tokens for these commands are

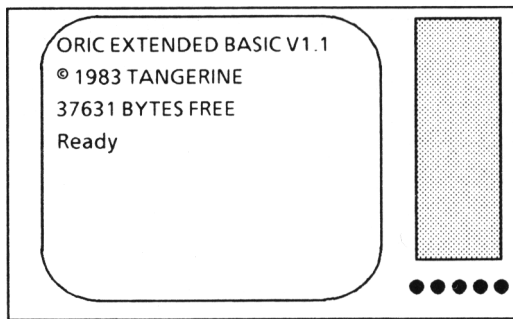
```
GOTO 151  
GOSUB 155  
ELSE 200  
THEN 201
```

CHAPTER 17

THE NEW OPERATING SYSTEM

With the release of an improved version of the ORIC's operating system, known as Version 1.1, the manufacturers have corrected many of the bugs we have mentioned in previous chapters, and added some extra commands to BASIC. This chapter describes the differences between the two operating systems, and explains how to use the extra facilities provided by Version 1.1 (V1.1).

The first thing you will notice when working with V1.1 is that on switching on, the display will look like this (on a 48K machine):



Notice that the number of bytes free now corresponds to the amount of memory available to BASIC.

NEW COMMANDS

Four of the five extra commands recognised by Version 1.1 are concerned in some way with

cassette operation. The first of these enables you to check that a program has been **CSAVEd** correctly, by searching through a program on tape, and verifying that it is the same as that currently stored in the ORIC. The command is a variation of **CLOAD**.

CLOAD "programe",V (,S)

where "programe" is the name of the program you wish to verify, and V stands for Verify. The ,S is optional to allow you to verify programs which have been **CSAVEd** at slow speed.

Use this command immediately after **CSAVE**ing a program to ensure that you have an accurate copy on the tape.

Type in the command, press **RETURN** and set the recorder to **PLAY**. The ORIC will search for the program you specified, indicating this with the message

Searching..

at the top of the screen.

If a program other than the one you specified is encountered, it's name will appear on the top line of the screen

Found PROGRAM1 B

and the ORIC will continue searching for the specified program.

Upon finding your program, the message will change to

Verifying .. programe B

and the ORIC will begin to verify. At the end of the process, the message

XX Verify errors detected

will appear on the screen. If XX is 0 then you have an accurate copy of your program on the tape, otherwise you'll have to **CSAVE** it again, perhaps at slow speed for a more certain save.

Another variant of **CLOAD** is

CLOAD "programe", J (,S)

where J stands for JOIN. This command allows you to load a program from tape and Join it to one which is already stored in the ORIC.

For example, suppose you had a program in the ORIC and wanted to renumber it (using the program in chapter 16). With the V1.0 Operating System, this isn't possible, but the Join command allows you to load the renumber program (or any other program or subroutine you have on tape) without disturbing the program already in the ORIC.

To be sure of success, neither program should have any line numbers in common and the program to join must start at a higher line number than the last line number of the program in the ORIC.

There are occasions where this doesn't matter, and the result of Joining two programs could give a sequence of line numbers like this

10
20
30
10
20

30

40

If there are no **GOTOs**, **GOSUBs** or other jumps, such a program will **RUN** correctly (though you would have problems **EDITing** it), but otherwise the ORIC will be confused.

V1.1 includes two other new commands to enable you to save data on tape in the form of arrays, and to reload it into a program.

STORE A, "name" (,S)

allows you to save an array on tape, with any name up to a maximum of 16 characters. The array can be Real (A()), Integer (A%()) or String (A\$()) and of any size within the limits of available memory, but must have previously been correctly **DIMensioned**.

This short program demonstrates the use of the **STORE** command, by saving an integer array on tape.

```
10  DIM A%(9,9)
20  FOR I = 1 TO 10
30  FOR J = 1 TO 10
40  A%(I,J) = I*J
50  NEXT J,I
60  CLS
70  PRINT"PRESS PLAY & RECORD ON
    TAPE DECK"
80  PRINT"AND PRESS ANY ORIC KEY"
90  GET A$
100 STORE A%,"Times table"
```

Whilst the array is being **STOREd**, the message

Saving...Times table I

will appear on the screen. The letter I indicates that an integer array is being saved, - this would be an S in the case of a string array, and an R for a Real array.

NOTE It is NOT POSSIBLE to use string variables as names for arrays, you must always specify the name in quotes within the program as in the example.

To allow you to recall such arrays, there is a further command

RECALL A, "name" (,S)

This will recall an array called "name" from the tape, and store it in array A. The array must previously have been **DIMensioned** correctly, and again, it isn't possible to use string variables as the name for the array. The following program illustrates the use of **RECALL**, by recalling the "Times table" array of the last program.

```
10 DIM A%(9,9)
20 RECALL A% "Times table"
```

Whilst the ORIC is searching for the integer array "Times table" the message

Searching..

is displayed on the top line of the screen. If any other arrays or programs are encountered on the tape their names are displayed as they are found

Found.. APROGRAM B

When the specified array is found, the message

Loading.. Times table I

will be displayed and the data will be loaded into the array A%.

When the data has been loaded, the message

Errors found

may be displayed. This indicates that errors were encountered during the loading of the data, and that the data **may** be suspect. This doesn't necessarily mean that the data in the array will be incorrect, just that you should treat the data with caution.

The final new command is used for printing data, strings or expressions at specified points on the screen.

PRINT @ X,Y;

where X and Y are coordinates on the TEXT or LORES screen. This command is very similar to **PLOT**, except that you cannot display inverse characters using **PRINT @**. As with **PRINT**, **PRINT @** can be abbreviated to ? @ when typing in programs.

NOTE The screen coordinate system used by V1.1 differs from that used by V1.0 in that column 0 is now the left hand ('protected') column, and is therefore not protected from **PRINT @**.

OTHER IMPROVEMENTS

In addition to the extra BASIC commands, V1.1 has tidied up many of the flaws in V1.0. Some of the more notable changes are as follows:

TAB

The **TAB** command now works correctly - lines such as

```
PRINT TAB(5)"ORIC"TAB(15)"HANDBOOK"
```

will move the cursor to the specified column on the screen before printing the message.

When **LISTing** a program, you can halt the listing by pressing the space bar, and then stop completely by pressing CTRL C, or continue by pressing the space bar

PRINT,

The screen is now divided into 5 columns of 8 character spaces and the use of a comma to separate items in a **PRINT** statement will cause each item to be printed starting at the next available column.

IF..THEN..ELSE

no longer gives inexplicable 'SYNTAX ERRORS'

STR\$

the spurious character preceding a string created with **STR\$** is now a space (CHR\$(32)). This means that such strings no longer appear green when **PRINTed**.

FILL

now leaves the HIREs cursor at the Y position after the last row to be FILLed.

You will also notice that the TEXT screen no longer scrolls downwards.

ADDRESS BOOK PROGRAM

The following program makes use of most of the additional features provided by V1.1 to allow you to use your ORIC as a flexible database

```
1      REM  ORIC ADDRESS BOOK
2      :
3      :
10     GOSUB 50000 ' SET UP

95     REM  FIRST MENU
96     :
100    CLS:PRINT@ 2,1;"ORIC ADDRESS
      BOOK"
110    PRINT@ 10,10;"Load file"
120    PRINT@ 10,15;"New file"
130    PRINT@ 10,22;"ESCAPE"

200    GET K$
210    IF K$ = "L" THEN GOSUB
      LOAD:GOSUB INSPECT:GOTO 100
220    IF K$="N" THEN GOSUB NW:GOSUB
      INSPECT:GOTO 100
230    IF K$ <> ESC$ THEN 200
240    CLS : END

1000   REM          INSPECT
1003   :
1005   REM          MAIN MENU
```

```
1010      :
1020      GOSUB HEADER
1030      PRINT@ 10,10;"Enter new data"
1040      PRINT@ 10,12;"Find item"
1050      PRINT@ 10,14;"List all items"
1060      PRINT@ 10,16;"Save file to
         tape"
1070      PRINT@ 10,22;"ESCAPE"

1100      GET K$
1110      IF K$ = "E" THEN GOSUB ENTER:
         GOTO INSPECT
1120      IF K$ = "F" THEN GOSUB FIND:
         GOTO INSPECT
1130      IF K$ = "L" THEN GOSUB LST:
         GOTO INSPECT
1140      IF K$ = "S" THEN GOSUB SAVE:
         GOTO INSPECT
1150      IF K$ <> ESC$ THEN 1100

1180      REM  CHECK FILE IS SAVED
1190      :
1200      GOSUB HEADER
1210      PRINT@ 5,10;ESC$"A DO YOU
         WANT TO SAVE THE FILE?"
1220      PRINT@ 10,14;"Yes"
1230      PRINT@ 10,16;"No"
1240      GET K$
1250      IF K$ = "Y" THEN GOSUB SAVE:
         GOTO INSPECT
1260      IF K$="N" THEN RETURN
1270      GOTO 1250

2000      REM          ENTER
2003      :
2005      REM  ADD NEW ITEMS
2010      :
2020      IF P < EL THEN 2100
2025      REM IF NO SPACE PRINT MESSAGE
```

```
2030  CLS
2040  PRINT@ 12,9;CHR$(4)ESC$"J
      INDEX FULL"CHR$(4)
2050  WAIT 300
2060  RETURN

2090  REM  ADD NEW ITEM
2100  GOSUB HEADER
2110  PRINT@ 5,5;"ENTER DATA"
2120  PRINT
2130  FOR I=0 TO FLDS
2140  PRINT F$(I);:INPUT A$(P,I)
2150  NEXT
2160  P = P+1

2190  REM  DISPLAY NEW ITEM
2195  :
2200  GOSUB HEADER
2210  ID = P-1
2220  GOSUB ITEMDISP
2230  PRINT@ 2,20;"Press RETURN to
      accept this item"
2240  PRINT" or DEL to delete it";
2240  GET K$
2250  IF K$ = CHR$(13) THEN RETURN
2260  IF K$ <> DEL$ THEN 2240
2270  PRINT
      CHR$(14)CHR$(11)CHR$(14)
2280  GOSUB DEL
2290  RETURN

3000  REM          FIND
3003  :
3005  REM  SEARCH FOR ITEM
3010  :
3020  GOSUB HEADER
3025  REM  ARE THERE ANY ENTRIES?
3030  IF P = 0 THEN 3700
```

```
3035  REM  IF NO ENTRIES PRINT
      MESSAGE
3040  PRINT:INPUT"FIND";FIND$
3050  PRINT:PRINT"in:";
3060  FOR I=0 TO FLDS
3070  PRINT TAB(10)I+1,F$(I)
3080  NEXT I
3100  GET K$
3110  IF ASC(K$) < 49 OR ASC(K$) >
      49+FLDS THEN 3100
3120  F = ASC(K$)-49
3130  CLS
3140  PRINT@ 2,1;"Searching.."

3190  REM  CLEAR FOUND ITEM POINTER
3200  FOR I=0 TO P
3210  FP(I) = -1
3220  NEXT
3230  FF = 0
3300  FOR I = 0 TO P
3310  IF A$(I,F)=FIND$ THEN
      FP(FF)=I:FF=FF+1
3320  NEXT I
3400  CLS:PRINT@ 2,1;FF" ITEMS
      FOUND"
3410  IF FF=0 THEN WAIT 200:RETURN

3490  REM  DISPLAY FOUND ITEMS
3500  FOR I = 0 TO FF-1
3510  ID = FP(I)
3520  GOSUB ITEMDISP
3530  PRINT:PRINT"Press N for next
      item or DEL to delete this
      one";
3550  GET K$
3560  IF K$="N" AND K$ <> DEL$ THEN
      3550
3570  PRINT CHR$(14) CHR$(11)
      CHR$(14)
3580  IF K$="N" THEN 3630
3590  GOSUB DEL
```

```
3600   FOR II = 0 TO FF-1
3610   FP(II) = FP(II)-1
3620   NEXT II
3630   NEXT I
3640   RETURN

3690   REM IF NO ITEMS PRINT MESSAGE
3700   PRINT@ 6,9;"CHR$(4)ESC$"J
      THERE ARE NO ITEMS TO
      FIND"CHR$(4)
3710   WAIT 300
3720   RETURN

4000   REM          LIST
4003   :
4005   REM    LIST ALL ITEMS
4010   :
4020   GOSUB HEADER
4030   PRINT@ 3,5;"LIST"
4035   REM FIRST CHECK ITEMS PRESENT
4040   IF P = 0 THEN 4300
4100   ID = 0

4105   REM DISPLAY ITEMS
4110   REPEAT
4120   GOSUB ITEMDISP
4130   PRINT:PRINT"Press N for next
      item,"
4140   PRINT"DEL to delete this one
      or ESC to end";
4150   GET K$
4160   PRINTCHR$(14)CHR$(11)CHR$(14)
4170   IF K$=DEL$ THEN GOSUB
      DEL:GOTO 4190
4180   ID = ID+1
4190   UNTIL K$ = ESC$ OR ID = P
4200   RETURN

4290   REM IF NO ITEMS PRINT MESSAGE
```

```
4300 PRINT@ 6,9;CHR$(4)ESC$"J
      THERE ARE NO ITEMS TO
      LIST"CHR$(4)
4310 WAIT 300
4320 RETURN
```

```
5000 REM          SAVE
5003 :
5005 REM  SAVE FILE TO TAPE
5010 :
5020 GOSUB HEADER
5030 PRINT@ 5,5;"SAVE"
5040 PRINT:PRINT:PRINT"Press PLAY
      and RECORD on tape deck"
5050 PRINT:PRINT"then any ORIC
      key"
5060 GET K$
5100 STORE A$,"ADDRESSES"
5110 PING
5200 PRINT:PRINT "Stop tape, then
      press:"
5210 PRINT:PRINT TAB(10)"C to
      continue"
5220 PRINT:PRINT TAB(10)"or ESC to
      end"
5300 GET K$
5310 IF K$="C"THEN RETURN
5320 IF K$=ESC$ THEN POP:RETURN
5330 GOTO 5300
```

```
10000 REM          LOAD
10003 :
10005 REM  LOAD FILE FROM TAPE
10010 :
10020 CLS:PRINT@ 2,1;"ORIC ADDRESS
      BOOK"
10030 PRINT@ 5,3;"LOAD FILE"
```

```
10040 PRINT:PRINT"Start tape deck"
10050 RECALL A$,"ADDRESSES"
10060 PING
10070 PRINT:PRINT"Stop tape then
press any key"
10080 GET K$

10085 REM COUNT ENTRIES
10090 P = -1
10100 REPEAT
10110 P = P+1
10120 UNTIL A$(P,0) = ""
10130 RETURN

11000 REM NEW FILE
11003 :
11005 REM BLANK OUT ARRAY
11010 :
11020 CLS:PRINT@2,1;"ORIC ADDRESS
BOOK"
11030 PRINT@ 5,3;"NEW FILE"
11040 FOR I=0 TO EL:FOR J=0 TO FLDS
11050 A$(I,J) = ""
11060 NEXT J,I
11070 P=0 'SET ITEM COUNT TO ZERO
11070 RETURN

20000 REM ITEM DISP
20003 :
20005 REM DISPLAY 1 ITEM
20010 :
20020 PRINT:PRINT A$(ID,0)
20030 FOR II = 1 TO FLDS
20040 PRINT:PRINT TAB(5)A$(ID,II)
20050 NEXT
20060 RETURN
```



```
21000 REM          DEL
21003 :
21005 REM  DELETE 1 ITEM
21010 :
21020 PRINT:PRINT"DELETING THIS
      ITEM"
21030 FOR II=ID TO P-1
21040 FOR JJ=0 TO FLDS
21050 A$(II,JJ) = A$(II+1,JJ)
21060 NEXT JJ,II
21070 PRINT"ITEM DELETED"
21080 P = P-1
21090 RETURN
```

```
22000 REM          HEADER
22010 :
22020 CLS
22030 PRINT@ 2,1;"ORIC ADDRESS
      BOOK"
22040 PRINT@ 25,1;P" ENTRIES"
22050 RETURN
```

```
50000 REM          SET-UP
50010 :
50020 EL = 100
50030 FLDS = 6
50040 F$(0) = "SURNAME"
50050 F$(1) = "FIRST NAME"
50060 F$(2) = "NUMBER & STREET"
50070 F$(3) = "TOWN"
50080 F$(4) = "COUNTY"
50090 F$(5) = "POST CODE"
50100 F$(6) = "PHONE NUMBER"
50110 DIM A$(EL,FLDS)
50120 DIM FP(EL)
```

```
50190  REM   SUBROUTINE ADDRESSES
50200  INSPECT = 1000
50210  ENTER   = 2000
50220  FIND    = 3000
50230  LST     = 4000
50240  SAVE    = 5000
50250  LOAD    = 10000
50260  NW      = 11000
50270  ITEM DISP = 20000
50280  DEL     = 21000
50290  HEADER  = 22000

50300  ESC$ = CHR$(27)
50310  DEL$ = CHR$(127)
50320  RETURN
```

The program is designed to create and maintain indexes, which may be saved and read back from tape using the new **STORE** and **RECALL** commands.

The listing given is for an address-book program, but may easily be modified to store catalogues or indexes of all sorts.

The program is controlled using *menus*. A list of options is displayed, and you select the one you want by pressing the appropriate key - usually the first letter of the option title. The ESC key is used to stop the program.

The program is written in *modules*. A main menu routine calls a number of subroutines which are independent of each other and perform different data-management tasks: entering new data, listing the data and so on.

PROGRAM DESCRIPTION

The first line of the program calls the subroutine at 50000 which sets up the array in which the data is stored and the other main variables of the program.

Lines 100 - 240 display an initial menu offering the options:

- Load a file from tape
- create a New file
- stop the program (ESCape)

Pressing N for New or L for Load carries out the selected task and then subroutine INSPECT is called.

Subroutine INSPECT (lines 1000 - 1270) is the heart of the program. A second menu is displayed, offering the options:

- Enter new data
- Find an item of data
- List all the items
- Save the file on tape
- return to the first menu (ESCape)

and the appropriate subroutine is called when an item is selected. If ESC is pressed the user is asked whether the file should be saved on the tape before the program returns to the first menu.

ENTER (2000) adds new data to the file.

FIND (3000) searches for specified items.

LST (4000) lists all the entries in the file.

SAVE (5000) saves the file on tape.

LOAD (10000) loads a file from the tape.

NW (11000) Blanks out the file and sets the entry counter (P) to zero.

ITEMDISP (20000) displays 1 entry of the file.

DEL (21000) deletes an entry.

HEADER (22000) prints a message at the top of the screen.

The file is stored in the array A\$, which is dimensioned in 50110. The other main variables used are:

P Pointer to the first free entry (and therefore also a count of the number of entries in the file)

EL The maximum number of entries

FLDS The number of 'fields' in each entry - the second dimension of A\$

F\$() The names of the fields

ID The item to be displayed by ITEMDISP

FF The number of found items in FIND

FP() Pointers to the found items

I, II, J and JJ are used as counters in loops.

To alter the program to store different information, change the names of the fields in lines 50040 onwards. If you alter the number of fields, change FLDS in line 5030 to the number of the last field. The name used as a file name when loading and

saving may be changed by altering lines 5100 and 10050.

If you want to add more features they may easily be 'plugged in'. For example, a routine to sort the entries into alphabetical order (perhaps based on the string sorting program in Chapter 7 - page 65) could be added as subroutine 6000 by adding two lines to INSPECT to display an extra menu item and call the subroutine. To make the routine fit neatly you could add a line

SORT = 6000

to subroutine 50000 and use the name SORT when calling the routine.

APPENDIX 1

BASIC COMMANDS

ABS (N)	Returns the absolute value of a number (removing the minus sign). See Chapter 8
AND	Logical operator. Returns the value TRUE if both operands are true. $A = B \text{ AND } C$ A is TRUE if B and C are both TRUE. Can also operate on binary values of numbers. Chapter 9
ASC (C\$)	Function. Returns the ASCII code of a character, or of the first character of a string. Chapter 7
ATN (N)	Function. Gives the angle in radians whose arctangent is N. Chapter 8
AUTO	Used with CSAVE when saving programs to cassette. Program will start running automatically when loaded. Examples: CSAVE "prog",AUTO CSAVE "prog",S,AUTO

Chapter 16

CALL N

Runs a machine code sub-routine at given address.

Chapter 15

CHAR C,S,FB

Puts characters at cursor position on HIRES screen. Parameters are:

C ASCII code for character (32 - 128)

S 0 - standard, 1 - alternate character set

FB Foreground / Background (See note below)

Chapter 13

CHR\$(N)

Converts ASCII codes to characters in string form.

Chapter 7

CIRCLE R,FB

Draws a circle on HIRES screen with centre at current cursor position. R is radius (1-99), FB is Foreground / background (see note).

Chapter 13

CLEAR

Clears all variables.

Chapter 15

CLOAD "prog"

Loads programs or data from tape. There are two variants:

CLOAD "prog" loads a program saved in 'fast' format.

CLOAD "prog",S loads a 'slow' program.

Chapter 14

CLOAD "prog",J Joins a program on tape to a program already in the ORIC. The program to be joined must have higher line numbers than that stored in the ORIC.
V1.1 only - Chapter 17

CLOAD "prog",V Verifies that a program on tape is the same as that currently stored in the ORIC.
V1.1 only - Chapter 17

CLS Clears screen.
Chapter 3

CONT Restarts program after a break. Only possible if no alterations have been made to the program.
Chapter 5

COS (N) Gives cosine of angle, which must be given in radians.
Chapter 8

CSAVE Saves programs to tape. There are two options:

CSAVE "PROG"
saves at fast speed
CSAVE "PROG" ,S
saves at slow speed.

See also AUTO.

Chapter 14

CURMOV X,Y,FB Moves the HIRES cursor. X and Y are the distances the cursor is to move horizontally and vertically. FB is Fore-ground / Background.
Chapter 13

CURSET X,Y,FB Positions the cursor on the HIRES screen. X and Y are absolute coordinates, FB is Foreground / Background.
Chapter 13

DATA Marks a list of string or numeric data written into a program. The items must be separated by commas.
Chapter 6

DEEK (LOC) Returns the number stored in the two bytes LOC and LOC + 1 as (contents of LOC) + 256*(contents of LOC + 1)
Chapter 15

DEF FNA(N) Defines a user - definable function.
Chapter 8

DEF USR = N Defines beginning of routine called by USR.
Chapter 15

DIM A(L,M) Dimensions arrays. Arrays of 1 dimension with up to 10 elements may be used without DIM.
Chapter 4

DOKE LOC,N	Stores the integer value of N in LOC and LOC + 1 Chapter 15
DRAW X,Y,FB	Draws lines on the HIRES screen from current cursor position. X and Y are the distance horizontally and vertically between the beginning and end of the line. Chapter 13
EDIT	Prints a program line on the screen with the cursor at the left-hand end ready for editing. Chapter
ELSE	See IF. Chapter 5
END	Ends program. Program may be restarted using CONT. Chapter 5
EXP (N)	Function returning exponential of N (e^N). Acts as natural antilog function. Chapter 8
EXPLODE	A preset noise. Chapter 10
FALSE	Preset variable containing 0. Useful in logical programming. Chapter 9
FILL X,Y,C	Fills X character positions on Y rows of the HIRES screen

- with the value C. Moves cursor down Y rows.
Chapter 13
- FNA (N)** Calls function defined by DEF FN.
Chapter 8
- FOR** Begins loop. E.g.
- FOR N = A TO B STEP C
- All lines as far as NEXT command are repeated with value of N increased each time from A to B in steps of C. STEP may be omitted, in which case variable is increased by 1.
- Chapter 5
- FRE (0)** Returns the amount of memory at present unused by BASIC.
Chapter 15
- FRE ("")** Forces garbage collection.
Chapter 15
- GET N** Halts program until a key is pressed. Stores key value in variable. The variable may be a string variable, in which case any key may be pressed, or a number variable, for which number keys only may be pressed.
Chapter 5

- GOSUB** Program branches to sub-routine at specified line, returning to instruction after GOSUB when RETURN is encountered.
Chapter 5
- GOTO** Program branches to a specified line.
Chapter 5
- GRAB** Allows BASIC to use the memory area normally reserved for the HIRES screen. HIRES may not be used until RELEASE command given.
Chapter 15
- HEX\$(N)** Converts a number to a string containing the characters of the hexadecimal representation of the number.
Chapter 8
- HIMEM** Sets the top of the memory area available to BASIC programs. May be used to create 'safe' area for storing machine code programs.
Chapter 15
- HIRES** Switches to high resolution graphics mode and clears screen to black. The HIRES cursor is set to 0,0.
Chapter 13

IF (condition) THEN (action) ELSE (action)	If the condition is true the action after THEN is carried out, otherwise the action after ELSE is performed. ELSE and its action may be omitted. Chapter 5 and Chapter 9
INK N	Sets the foreground colour. N must be a number between 0 and 7. Chapter 3
INPUT N INPUT "data";N	Prompts operator for an input and stores it in variable. Variable may be a number or string, the input data must correspond. Chapter 4
INT (N)	Returns integer component of real number. E.g.: INT (3.75) returns 3. Chapter 8
KEY\$	Reads keyboard, but does not wait for key to be pressed. Key character must be stored in string variable: A\$ = KEY\$ A\$ will contain character value of the last key pressed, if any key has been pressed since KEY\$ was last used. Chapter 6

LEFT\$(C\$,N)	Returns the first N characters of the string. Chapter 7
LEN(C\$)	Gives the number of characters in the string. Chapter 7
LET	Optional. May be used when assigning values to variables: <pre>LET P = 5 and P = 5</pre> have the same effect.
LIST	Lists the specified lines of the program on to the screen Chapter 4
LLIST	Lists the specified lines to the printer.
LN(N)	Returns the natural logarithm of a number. Chapter 8
LOG(N)	Returns the base 10 logarithm of a number. Chapter 8
LORESS	Sets low resolution display mode. S specifies character set: 0 for standard, 1 for alternate. Chapter 12
LPRINT	Sends data to a printer.

MID\$ (C\$,I,N)	Returns N characters of string beginning at the Ith character. Chapter 7
MUSIC CH,O,N,V	Plays Nth note of Oth octave on channel CH at volume V. Chapter 10
NEW	Clears a program and its variables from memory. Chapter 4
NEXT N	Marks end of loop begun by FOR. The variable need not be specified. Chapter 5
NOT	Logical operator. Reverses truth of expression (e.g. NOT TRUE returns FALSE). Can also be applied to binary values of numbers. Chapter 9
ON N GOSUB	Program branches to Nth subroutine in list. If N is larger than number of items in list no branch occurs. Chapter 5
ON N GOTO	Program branches to Nth destination in list. If N larger than number of items in list no branch occurs. Chapter 5
OR	Logical operator. Returns value TRUE if either or both operands are true. E.g.:

A = B OR C

A is true if B or C is true.
Can also act on binary values
of numbers.

Chapter 9

PAPER N

Sets background colour of
display. N must be 0 - 7.

Chapter 3

PATTERN N

Sets the line pattern in
HIRES graphics. N must be
0 - 255.

Chapter 13

PEEK (LOC)

Gives the contents of memory
location LOC.

Chapter 15

PI

Preset variable containing π
(3.14159265).

Chapter 8

PING

Preset sound.

Chapter 10

PLAY CH,N,E,P

Controls sound production in
conjunction with SOUND
and MUSIC commands.

Chapter 10

PLOT X,Y,C\$

Places a character or string
on the TEXT or LORES
screen at position specified by
X and Y. The ASCII code of a
character may be used
instead of a string.

Chapter 4

POINT (X,Y)	Returns 0 if the pixel at X,Y on the HIRES screen is background and -1 if it is foreground. Chapter 13
POKE LOC,N	Puts value of N into memory location LOC. N must have value 0 - 255. Chapter 15
POP	Removes RETURN address from the stack of RETURN addresses. The next RETURN statement to be executed will cause a return to the instruction after the second most recent GOSUB. Chapter 15
POS (0)	Gives the present horizontal position of the cursor on the TEXT and LORES displays. The number in brackets has no significance. Chapter 4
PRINT	Puts data, numbers or characters on the screen. May be written as "?". Chapter 3
PRINT @ X,Y;	Prints data or characters at position X,Y on the screen. Can be abbreviated to ? @. <i>V1.1 only - Chapter 17</i>
PULL	Similar to POP, but removes one address from the stack in REPEAT ... UNTIL loops. Chapter 15

READ	Copies items from DATA statements into variables. Chapter 6
RECALL A,"name"	Recalls a previously dimensioned array A, which may be Real, Integer or String, from tape. <i>V1.1 only - Chapter 17</i>
RELEASE	Prevents BASIC using the memory area of the HIRES display. Allows HIRES to be used. See also GRAB. Chapter 15
REM	Allows remarks to be inserted in programs as an aid to readability. Remarks are ignored when program is run. Chapter 4
REPEAT	Marks beginning of loop, the end of which is marked by UNTIL. Loop is repeatedly executed until condition after UNTIL is satisfied. Chapter 5
RESTORE	Returns READ pointer to first DATA item. Chapter 6
RETURN	Marks end of subroutine. Program returns to the instruction after the GOSUB instruction which called the subroutine. Chapter 5

RIGHT\$(C\$,N)	Returns the N rightmost characters of string. Chapter 7
RND (N)	If $N > 0$, returns random number. If $N = 0$, repeats last number. If $N < 0$, reseeds random number generator. Chapter 8
RUN	Begins execution of BASIC program. All variables are cleared. A line number may be given, otherwise execution begins at the lowest numbered line. Chapter 4
SCRN (X,Y)	Gives the ASCII code of the character at position X,Y of the TEXT or LORES display. Chapter 12
SGN (N)	Returns 1, 0 or -1 according to whether number is positive, zero or negative. Chapter 8
SOUND CH,P,V	Produces a sound of period P and volume V on channel CH. Chapter 10
SPC (N)	Prints N spaces on the screen. N must be 0-255. Chapter 4

SQR (N)	Returns the square root of the argument. Chapter 8
STOP	Stops execution of program. A message 'BREAK IN xxx' is printed. Program may be restarted using CONT. Chapter 4
STORE A,"name"	Saves an array A, which may be Real, Integer or String, on tape. <i>V1.1 only - Chapter 17</i>
STR\$ (N)	Converts numbers to strings of numeric characters. Chapter 7
TAB (N)	Moves the cursor N places from the left edge of the screen in a PRINT statement. Chapter 4
TAN (N)	Gives the tangent of an angle, which must be in radians. Chapter 8
TEXT	Switches to TEXT display mode. Has little effect when used in LORES: CLS works better! Chapter 13
THEN	See IF. Chapter 5
TO	See FOR. Chapter 5

TROFF	Switches off trace. Must be used in program. Chapter 15
TRON	Switches on trace. Must be used within program. When program is run, all line numbers executed after TRON are printed on the screen. Chapter 15
TRUE	Preset variable containing -1. Useful in logical arithmetic. Chapter 9
UNTIL (condition)	Marks end of loop begun by REPEAT. Loop repeats until condition after UNTIL is satisfied. Chapter 5
USR (N)	Calls machine code routine at address previously specified by DEF USR. Value in brackets is placed in floating point accumulator. Returns contents of floating point accumulator after subroutine is run. Chapter 15
VAL (C\$)	Converts string of number characters to number. Chapter 7
WAIT N	Program pauses for period of N times 1/100 second. Chapter 5

ZAP

Preset noise.
Chapter 10

NOTE

FB is used to represent the foreground / background parameter in high resolution commands. This parameter has the following effect:

FB = 0 Draw in
 background
 colour.

FB = 1 Draw in
 foreground colour.

FB = 2 Draw in inverse of
 colour on screen
 already.

FB = 3 Draw nothing.

APPENDIX 2

BASIC ERROR MESSAGES.

If the ORIC encounters a command which it is unable to execute, or a number it cannot handle, an error message will be displayed.

If the error occurred whilst running a program, the program will stop, and a message of the form

***** ERROR IN XXXX

will be displayed, where ***** represents the type of error, and XXXX is the program line in which the error has occurred. The program is retained in the computer, as are the values assigned to all variables at the time of the error.

In Immediate mode, the error message takes the form

***** ERROR

The following descriptions explain the error messages, and the possible reasons for them.

CAN'T CONTINUE

You may not continue the RUNning of a program, using the command CONT, if the program has been edited in any way.

DISP TYPE MISMATCH ERROR

You may not use the following HIRES commands in LORES or TEXT modes:

**CURMOV
CURSET
DRAW
CIRCLE
PATTERN
POINT
CHAR**

Similarly, **PLOT** and **SCRN** are not allowed in HIRES mode.

DIVISION BY ZERO

It is impossible to divide by zero.

FORMULA TOO COMPLEX

A series of string handling functions are too complex to be carried out in one step. They must be split into smaller steps.

ILLEGAL DIRECT

The following BASIC commands may not be used in Immediate mode:

**DEF FN
GET
INPUT
TROFF
TRON**

ILLEGAL QUANTITY

A parameter passed to a **Sound** or **Graphics** command, mathematical or string function was outside the allowed range for that parameter.

NEXT WITHOUT FOR

The variable in a **NEXT** statement has no corresponding **FOR** statement.

OUT OF DATA

This error occurs when a **READ** command is executed, but there is either no **DATA**, or all the **DATA** has previously been read. The program either tried to read too much data, or there was insufficient data in the **DATA** statement.

OUT OF MEMORY

Either the program is too large, or you have used too many variables, too many **FOR ... NEXT** loops, too many **GOSUBs**, or you have allocated too much space for arrays with the **DIM** command.

OVERFLOW

The result of a calculation, or a directly entered number, was larger than 1.70141 *E38. If an underflow occurs, the result is given as zero, and *no* error message is displayed.

REDIM'D ARRAY

After an array was **DIMensioned**, a statement **DIMensioning** the same array was encountered.

RETURN WITHOUT GOSUB

An attempt has been made to execute a **RETURN** statement which was not preceded by a **GOSUB** statement.

STRING TOO LONG

A string has been concatenated(i.e. made up from several smaller strings joined together) and its length has exceeded 255 characters.

BAD SUBSCRIPT

An array element outside the dimensions of that array has been accessed, or the wrong number of dimensions has been used, e.g.

`Z(9,9,9) = A`, when Z is a two-dimensional array.

SYNTAX ERROR

This can be caused by :

- Use of a non-existent (or misspelled) BASIC command
- Missing brackets
- Incorrect punctuation
- Missing parameters
- Illegal characters

TYPE MISMATCH

An attempt to assign a numeric value to a string variable or vice-versa has occurred. A numeric argument was passed to a function requiring a string argument or vice-versa. For example:

```
PRINT ASC(A) instead of
PRINT ASC("A")
```

A=Z\$
or A\$=2

UNDEF'D STATEMENT

The program attempted to **ELSE**, **GOTO**, **GOSUB**, **THEN** or **RUN** a non-existent statement.

UNDEF'D FUNCTION

An attempt to reference an undefined user defined function has been made.

REDO FROM START

A string function has been entered when the program was executing an **INPUT** statement. The message will be repeated until a number is entered.

BAD UNTIL

An **UNTIL** statement has been encountered without a corresponding **REPEAT** statement being executed.

APPENDIX 3

ATTRIBUTES

FOREGROUND BLACK	0	@	BACKGROUND BLACK	16	P
RED	1	A	RED	17	Q
GREEN	2	B	GREEN	18	R
YELLOW	3	C	YELLOW	19	S
BLUE	4	D	BLUE	20	T
MAGENTA	5	E	MAGENTA	21	U
CYAN	6	F	CYAN	22	V
WHITE	7	G	WHITE	23	W
SH / ST STANDARD	8	H	TEXT 60 HZ	24	X
SH / ST ALTERNATE	9	I	TEXT 60 HZ	25	Y
DH / ST STANDARD	10	J	TEXT 50 HZ	26	Z
DH / ST ALTERNATE	11	K	TEXT 50 HZ	27	}
SH / FL STANDARD	12	L	GRAPHICS 60 HZ	28	
SH / FL ALTERNATE	13	M	GRAPHICS 60 HZ	29	{
DH / FL STANDARD	14	N	GRAPHICS 50 HZ	30	~
DH / FL ALTERNATE	15	O	GRAPHICS 50 HZ	31	←

SH = Single height characters

DH = Double height characters

ST = Steady

FL = Flashing

NOTE The 60 HZ attributes are for the U.S.A. only - their use with U.K. Televisions may cause an unstable display.

APPENDIX 4

ASCII CODES

CODE	CHAR	CODE	CHAR	CODE	CHAR
32	space	64	@	96	©
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	\$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(72	H	104	h
41)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[123	{
60	<	92	\	124	
61	=	93]	125	}
62	>	94	↑	126	checks
63	?	95	£	127	delete

APPENDIX 5

CONTROL CHARACTERS

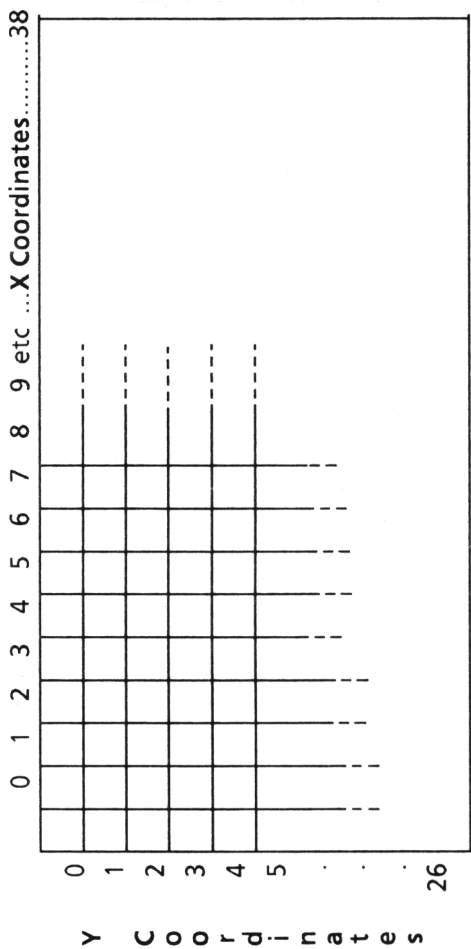
These characters can be used directly from the keyboard, or within programs, to control various functions of the ORIC. To use from the keyboard, hold down CTRL whilst pressing the appropriate key from the first column. To use in a program, use the format

`PRINT CHR$(N)`

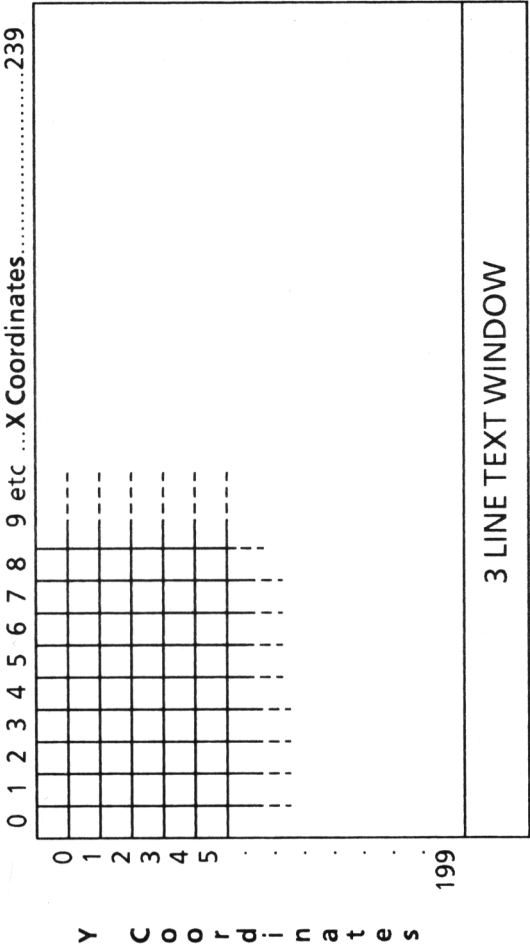
where N is the code given below in the second column.

CTRL	CODE	FUNCTION
A	N/A	Copies characters into keyboard buffer
C	N/A	Stops running of programs
D	4	Toggle auto double height on/off
F	6	Toggle keyclick on/off
G	7	'Bell'
H	8	Cursor left
I	9	Cursor right
J	10	Cursor down
K	11	Cursor up
L	12	Clear TEXT screen
M	13	Carriage return
N	14	Clear row
O	15	Turn off display until Carriage return
P	16	Toggle printer on/off
Q	17	Toggle cursor on/off
S	19	Toggle output to screen on/off
T	20	Toggle CAPS lock on/off
X	N/A	Clear line
Z	26	'ESC' character
[27	'ESC' character
]	29	Toggle protected column on/off
N/A	30	Cursor to home position

SCREEN MEMORY MAPS



Screen Map for TEXT and LORES modes



Screen Map for HIRES mode

APPENDIX 7

SYSTEM MEMORY MAP

ADDRESS		MODE	
DECIMAL	HEX	TEXT	HIRES
65535	FFFF	Basic Interpreter & Operating System in ROM	
49152	C000	Unused	Unused
49120	BFE0	TEXT / LORES Screen	HIRES Screen
48000	BB80	Alternate Character set	
47104	B800	Standard Character set	
46080	B400	User programs if GRAB command has been used	
40960	A000		Character sets
39936	9C00		
38912	9800	User programs	User programs
1280	500	Page 4 (400-420 for m/c)	
1024	400	Page 3 Physical I/O addresses	
768	300	Page 2 Run-time variables	
512	200	Page 1 Stack	
256	100	Page 0 System variables	

APPENDIX 8

NUMBERING SYSTEMS

Computers store and operate upon numbers in a different way from humans - they use a numbering system known as **Binary Notation**.

Binary notation is a means of representing quantities with groups of 1s and 0s. Humans use a system called **Decimal Notation**, in which quantities can be represented by combinations of up to ten symbols (the numbers 0 to 9).

Computers use the binary system because they are only able to recognise and differentiate between two states - ON and OFF. These two states can conveniently be represented by 1 (ON) and 0 (OFF). A single 1 or 0 is called a **Binary digit**, or **BIT**, and the **ORIC** stores data in the form of groups of eight of these bits, known as **BYTES**.

In the computer memory, one memory location is able to store one **Byte** of data (eight bits). A collection of 1024 of these bytes is called a **KILOBYTE**, or **k** for short. We can get an idea of the data storage capacity of a computer from the number of **k** of memory it has (48k, for example, is $48 * 1024 * 8$ bits).

We have described a byte as a collection of eight bits, like this:

11111111

This is an 8-bit binary number, which represents 255 in decimal notation. To see how this is so, we must first examine the decimal number and see what it means.

H	T	U
2	5	5

means

$$2*100 + 5*10 + 5*1$$

In other words, each digit is worth 10 times the one to its right.

Binary notation uses this same 'place value' principle, except each bit in a binary number is worth **double** that to its right. We can assign values to the eight bits in the same way as the 'Hundreds, Tens and Units' assigned to the digits of a decimal number.

128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1

By adding up, we can see why this number represents 255

1	*	128	
1	*	64	
1	*	32	
1	*	16	
1	*	8	
1	*	4	
1	*	2	
1	*	1	+
<hr/>			
		255	

You'll notice that 255 is the biggest number we can represent with an 8-bit binary number. Hence this is the largest number we can store in a single memory location.

As a further example, let's take the decimal number 170. To find its binary representation, we continuously divide by two, and the remainder becomes a bit in the binary number.

170/2	=	85	remainder	0
85/2	=	42	"	1
42/2	=	21	"	0
21/2	=	10	"	1
10/2	=	5	"	0
5/2	=	2	"	1
2/2	=	1	"	0
1/2	=	0	"	1

giving us the binary number

10101010 (reading upwards)

ADDITION OF BINARY NUMBERS

Binary numbers can be added together in the same way as decimal numbers. An example will make this clear.

To perform the sum

$$\begin{array}{r} 105 \\ + 19 \\ \hline 124 \end{array}$$

we add up the digits in each column to form that digit of the answer. If the result of this addition is greater than 9, we generate a carry into the next column. This principle also applies to binary numbers. Let's perform the same calculation with the binary forms of 105 and 19:

$$\begin{array}{r}
 01100100 \\
 11001110 \\
 \hline
 1 \quad 00110010
 \end{array}$$

The result is binary 50 - the method worked.

Notice that a carry was generated, indicating that the answer is positive. A consequence of using bit 7 as a sign bit is that the range of numbers we can represent with an 8-bit number is restricted to

-128 to +127

HEXADECIMAL

Manipulating numbers in binary is a lot easier for a computer than it is for a human, and one way in which binary numbers can be made more digestible is by representing them in hexadecimal notation, or **hex**.

Hex is a system of counting in base 16, using the symbols 0 to 9 and A to F as follows

DECIMAL	0	1...	9	10	11...	15	16	17
HEX	0	1...	9	A	B...	F	10	11

Thus FF (hex) represents 255 (decimal) and 11111111(binary).

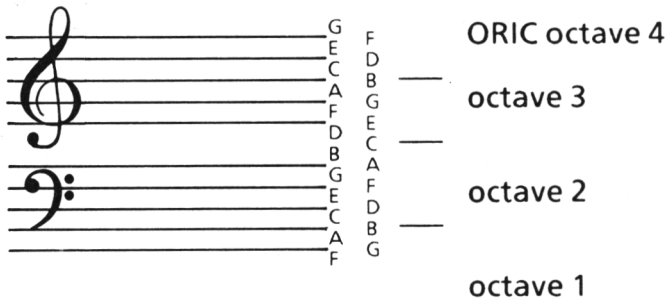
You will frequently encounter references to Hex, usually as memory addresses, because it is so convenient. (Which of these is easiest to recognise? 1111111111111111 or 65535 or #FFFF).

APPENDIX 9

MUSICAL NOTATION

This appendix will not teach you all about music, but it contains the basic information you need to translate sheet music into ORIC programs.

Music is written by positioning symbols which represent the length of notes on a framework (called a staff) representing the pitch.



The lengths of notes are indicated by the note shape:

A Semibreve

a Minim

a Crotchet

a Quaver

a Semiquaver



a Demisemiquaver

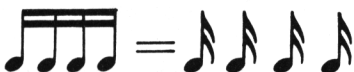


is twice as long as
which is twice as
long as
which is twice as
long as
which is twice as
long as
which is twice as
long as

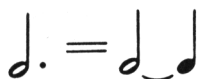
a Hemidemisemiquaver




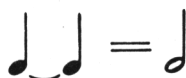
Tails on notes may go up  or down . The feathers on quavers and shorter notes may be joined where they appear in groups:



A dot after a note means that it is made half as long again as a normal note:



The mark  is a tie which means the notes are joined together.



Volume is indicated by markings below the stave.

ff

Very loud

f

Loud

mf

Moderately loud

mp

Moderately soft

p

Soft

pp

Very soft



Get louder



Get softer

Speed is indicated by markings which may be above or below the stave. Examples are:

Presto	which means Fast
Allegro	Quite fast
Allegro moderato	Moderately fast
Moderato	Medium pace
Andante	Smoothly
Largo	Very slow

Unfortunately, there are many other Italian words and phrases which may be used. The best thing to do is to adjust your ORIC program until the speed sounds right, and not worry too much about what is written on the music.

Two other markings which may appear next to notes are \sharp and \flat . \sharp (sharp) means that the note should be raised by one semitone (i.e. the ORIC note value should be increased by 1). \flat (flat) means that the note is lowered one semitone (the ORIC note value is decreased by 1).

All other markings which may appear on sheet music (and there are many of them) can be ignored.

LAND OF HOPE AND GLORY

Here is the beginning of 'Land of Hope and Glory', the example used in the program in Chapter 10.



The notes shown in this music correspond to the notes represented by the first number in each group of four in the DATA statements of the program. The length of the note indicated by its shape corresponds to the fourth number of each group.

Note the 'sharp' symbols (\sharp) on the lines of the stave corresponding to the note F. These mean that all F's throughout the music are sharpened.

SPEEDING UP PROGRAMS

There are several things you can do to increase the running speed of your BASIC programs. Unfortunately, this is usually at the expense of clarity, and you may find it useful to keep a 'Slow', but easy-to-follow version of your program, should you wish to amend it at a later date.

- 1 Remove all unnecessary spaces, REMs and indentation from the program. A small speed increase will result, because BASIC will not have to skip over redundant spaces to find executable commands.
- 2 Always use variables instead of constants. The ORIC can handle variables much more rapidly than numbers. This is especially important in REPEAT ... UNTIL and FOR ... NEXT loops.
- 3 Use as many statements per line (separated by ':') as possible.
- 4 Reuse the same variables whenever possible.
- 5 Use the zero elements of arrays when possible.
- 6 Assign often used variables early on in the program. The ORIC stores all variables in a table. The first declared variables are the first in the table and are found more quickly.

- 7 Put all subroutines near the start of the program. The ORIC searches through the whole program for a subroutine each time a **GOSUB** command is executed, and subroutines having low line numbers will be found and executed more quickly than those at the end of the program.
- 8 Omit the variable after **NEXT** in **FOR...NEXT** loops.
- 9 In programs where the keyboard is not often used, the routines which check for a key press can be disabled within a program with a line :

```
100 CALL #E6CA
```

and enabled each time you want to use **INPUT** or **GET** by

```
200 CALL #E804
```

Remember to include an enable statement before the program ends, to allow you to regain control of the computer.

APPENDIX 11

EDITING BASIC PROGRAMS

For some corrections to programs, the simplest thing to do is to retype the entire line. There is, however, a BASIC command to help you make corrections:

EDIT N

where N is a program line number. This command will list the appropriate line and place the cursor at the start of the line. To edit the line you can move the cursor along using CTRL A. The cursor will move along the line, copying each character into the input buffer, just as if you were retyping the line. When the cursor reaches the point at which you want to make a correction, release the keys and type. You must continue CTRL A-ing the rest of the line into the buffer after the correction before pressing RETURN to tell the ORIC you have finished editing that line.

NOTE if you move the cursor using the cursor keys, nothing will be copied into the buffer, so you can move the cursor to a clear part of the screen to type in sections to be added to the middle of a line.

EXAMPLE

Suppose you wanted to insert the command PING in the line

```
100 FOR I = 1 TO 10:PRINT I:  
    WAIT10: NEXT I
```

so that the ORIC **PINGed** for each value of I.
First, you would type

```
EDIT 100
```

Next, using CTRL and A, move the cursor along the line copying everything up to the end of the **PRINT I** command.

To enter the extra command, you would move the cursor to a clear part of the screen (using the cursor keys), type in the extra command and return the cursor to the start of the **WAIT** command. Because you want to retain the rest of the line, you would now CTRL A the remaining characters into the buffer, pressing RETURN at the end of the line. The line has now been edited.

NOTE If you change your mind or get confused half way through editing a line pressing CTRL X will abandon the **EDIT** and leave the line as it was.

INDEX

ABS	69	EDIT	246
AND	79	ELSE	41
Arrays	24	END	46
Arithmetic	11	Error messages	224
Assembly Language	161	ESC key	118
ASCII code	103, 230	EXP	72
ATN	71	EXPLODE	14, 86
Attributes	112, 229	FALSE	78
BASIC	1	FILL	141, 195
Commands	207	FN	75
Binary numbers	235	FOR	35
Bit	235	FRE	158
Boolean algebra	79	Functions	69
Branching	39	GET	50
Buffer	51	GOSUB	44
Byte	235	GOTO	39
CALL	161	GRAB	159
Cassette recorder	4, 148	Graphics	
Channels	87	High resolution	131
CHAR	138	Low resolution	120
Character sets	119	Character set	120
CHR\$	62	Hexadecimal numbers	239
CIRCLE	143	HEX\$	213
CLEAR	158	HIMEM	163
CLOAD	149, 189	HIRES	131
CLS	11	IF ... THEN ... ELSE	41
Colour	13, 112	Immediate mode	9
CONT	46	INK	13
COS	71	INPUT	27
CSAVE	148	INT	70
CURMOV	134	Integer variables	23
CURSET	135	Joining programs	190
Cursor	9, 132	Jumps	39
DATA	52	KEY\$	51
DEEK	160	Key click	9, 85
DEF FN	75	Languages	1
DEF USR	162	LEFT\$	60
DIM	26	LEN	59
DOKE	160	LET	215
DRAW	132	LIST	17
		LLIST	215
		LN	72

Loading programs	149	Reset button	162
LOG	72	RESTORE	53
Loops	35	RETURN	44
LORES	123	RIGHT\$	60
Lower case	231	RND	63
LPRINT	215	RUN	17
Machine code	161	Saving programs	148
Memory maps	232, 234	Screen	5, 131, 232
Memory use	156, 185	SCRN	220
Microprocessor	161	SGN	70
MID\$	60	SIN	71
Monitor	7	SOUND	80
Music	85, 240	SPC	55
MUSIC	95	SQR	69
		STOP	46
NEW	18	Stopping programs	39, 46
NEXT	35	STORE	191
NOT	80	String variables	23
Numbering systems	235	STR\$	63, 194
		Structured programming	163
ON ... GOSUB	47	Subroutines	44
ON ... GOTO	47	Syntax errors	9, 224
OR	80		
		TAB	55
PAPER	13	TAN	71
PATTERN	143	Tape	148
PEEK	159	TEXT	132
PI	71	THEN	41
PING	14, 86	TO	35
PLAY	91	Trace facility	154
PLOT	57	TROFF	154
POINT	135	TRON	154
POKE	160	TRUE	78
POP	155	Television	3
POS	55		
PRINT	9, 53, 194	UNTIL	32
PRINT @	193	Upper case	231
Program	16	User defined functions	75
PULL	155	User defined characters	107
		USR	162
Random numbers	73		
READ	52	VAL	63
Real variables	22	Variables	20
RECALL	192	Verifying programs	189
RELEASE	159		
REM	29	WAIT	40
REPEAT	32		
Reserved words	22	ZAP	14, 86

Available in Century's Science & Technology series

Dictionary of New Information Technology

A. J. MEADOWS, M. GORDON and A. SINGLETON

What to Buy for Business: A Handbook of New Office Technology

JOHN DERRICK and PHILLIP OPPENHEIM

The Electronic Mail Handbook

STEPHEN CONNELL and IAN A. GILBRAITH

The Way the New Technology Works

KEN MARSH

Mind

DAVID A. TAYLOR

Computers and Your Child

RAY HAMMOND

International Dictionary of Graphic Symbols

JOEL ARNSTEIN

In association with 'Personal Computer World'

Microcomputing for Business: A User's Guide

edited by DICK OLNEY

The Microcomputer Handbook: A Buyer's Guide

edited by DICK OLNEY

The Spectrum Handbook

TIM LANGDELL

The Intimate Machine

NEIL FRUDE

35 Educational Programs for the BBC Micro

IAN MURRAY

Educational Programs for the Dragon 32

IAN MURRAY

Educational Programs for the Spectrum

IAN MURRAY

Century Computer Programming Course

PETER MORSE and IAN ADAMSON

Information Technology Yearbook

edited by PHILIP HILLS

The Database Primer

ROSE DEAKIN

Computer Gamesmanship

DAVID LEVY

The Oric Handbook offers its readers the opportunity of harnessing the power of one of the newest and most exciting microcomputers. Peter Lupton and Frazer Robinson are both professional technical authors and their clear step-by-step introduction opens the Oric to the beginner while the wealth of hints and tips, exciting programs and applications makes the book essential reading for even the experienced programmer. There are also lengthy sections on sound and graphic capabilities which are particularly impressive features of the machine.

The book contains full details of the differences between the old and new ORIC ROM and also fully documents all the new features and commands.

The Oric Handbook is a must for anyone who wants to take their Oric to the limits – and beyond!

THE ART OF HAND BOOK UPON & ROBINSON

